



# 软件架构设计

温昱 著



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

# 软件架构设计

贯穿案例 实战性强 脉络清晰 整体性强  
方法主流 适用性强 深入浅出 层次性强



本书紧紧围绕“软件架构设计”这一主题，立足实践，解析了软件架构的概念，阐述了切实可行的软件架构设计方法，提供了可操作性极强的完整的架构设计过程。另外，本书从思维方式的突破、面向对象设计、UML建模、过程与管理等关键过渡环节，为广大程序员的成长提供了切中肯綮的指导。

全书共26章，分为三个部分：软件架构概念与思想篇、软件架构设计方法与过程篇、程序员成长篇。

理论与实践并重是本书的特点。架构设计要如何开展？架构设计要进行到什么程度？各类需求对架构设计的影响有何不同？关键需求决定架构的具体做法是什么？如何运用“属性-场景-决策”表规划非功能需求？如何运用OO原则进行敏捷设计？对这些问题本书都进行了深入阐述，并结合金融、航空、网络管理等行业软件的成功架构设计案例，将理性的思考和宝贵的实践经验奉献给了读者。

本书可作为计算机软件专业本科生、研究生和软件工程硕士的软件架构设计教材，也可作为软件开发高级培训、软件开发管理培训的培训教材，更是一线高级开发人员和开发管理人员的必备参考书。

## 专家推荐

“作者不但理论知识丰富，阅读广泛深入，设计工具运用自如，而且具有务实精神。书中举了大量来自开发一线的实际案例，用这些实际案例当中的设计决策来印证和讲述理论。当实践与理论发生矛盾的时候，他也是从实践出发，认真地分析原因，找出解决之道。……温昱这本书读起来却津津有味，不但能够澄清很多观念，学到不少知识，而且阅读过程中有时能进入一种状态，似乎跟着作者一起思考，甚至跟他发生争论。我想这正是因为作者的务实精神。……我乐于向希望学习软件架构设计的开发者和架构师们推荐这本务实之作。”

——孟岩 《程序员》杂志社技术主编

## 作者简介



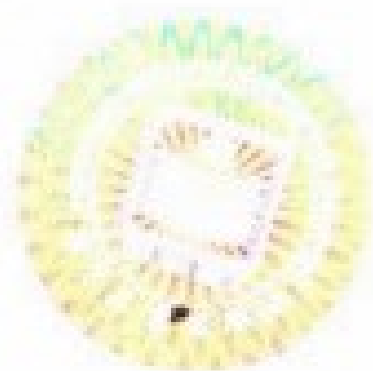
温昱 资深咨询顾问，CSAI特聘高级顾问，软件架构专家，软件架构思想的传播者和积极推动者。十年系统规划、架构设计和研发管理经验，在金融、航空、多媒体、网络管理、中间件平台等领域负责和参与多个大型系统的规划、设计、开发与管理。在《程序员》杂志、IBM DeveloperWorks 等媒体发表了《图论思想与UML应用》、《敏捷设计从理论到实践》、《随需而变的RUP》等文章数十篇。译著有《应用框架的设计与实现——.NET平台》等。松耦合空间([www.ou-he.com](http://www.ou-he.com))网站创办人。

图书分类：软件工程

网上订购：[www.dearbook.com.cn](http://www.dearbook.com.cn)  
第二书店·第一服务



责任编辑：周 筠 梁 晶  
责任美编：方 舟



本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

ISBN 978-7-121-03946-1



9 787121 039461 >

定价：45.00 元

# 软件架构设计

温 昱 著

电子工业出版社

Publishing House of Electronics Industry

北京 • BEIJING

数字图书馆  
PDG

## 内 容 简 介

本书紧紧围绕“软件架构设计”这一主题,立足实践解析了软件架构的概念,阐述了切实可行的软件架构设计方法,提供了可操作性极强的完整的架构设计过程。另外,本书从思维方式的突破、面向对象设计、UML 建模、过程与管理等关键过渡环节,为广大程序员的成长提供了切中肯綮的指导。本书可作为计算机软件专业本科生、研究生和软件工程硕士的软件架构设计教材,也可作为软件开发高级培训、软件开发管理培训的培训教材,更是第一线高级开发人员和开发管理人员的必备参考书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

### 图书在版编目(CIP)数据

软件架构设计 / 温昱著. —北京: 电子工业出版社, 2007.5

ISBN 978-7-121-03946-1

I. 软… II. 温… III. 软件设计—教材 IV. TP311.5

中国版本图书馆 CIP 数据核字 (2006) 第 103712 号

责任编辑: 周筠 梁晶

印 刷: 北京智力达印刷有限公司

装 订: 北京中新伟业印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

经 销: 各地新华书店

开 本: 787×1092 1/16 印张: 22.75 字数: 460 千字

印 次: 2007 年 5 月第 1 次印刷

印 数: 5 000 册 定价: 45.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

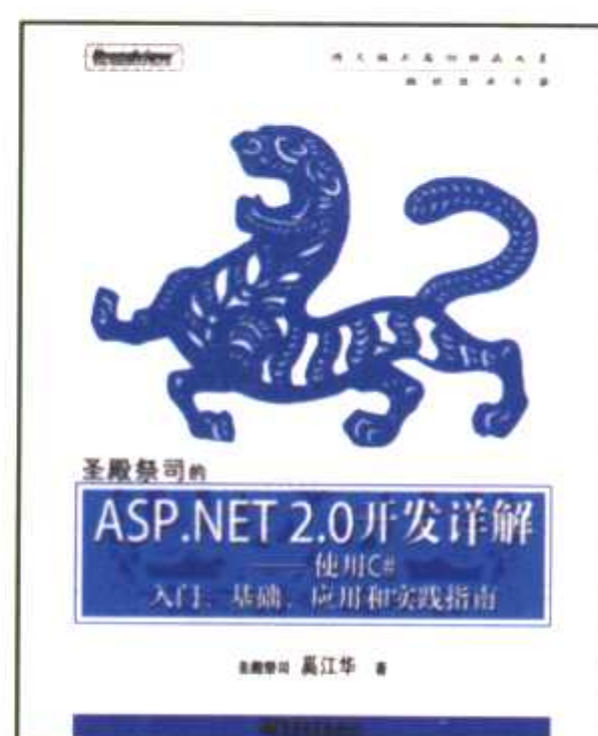




《SQL Server 2005 数据库开发详解》

胡百敬 姚巧玫 著

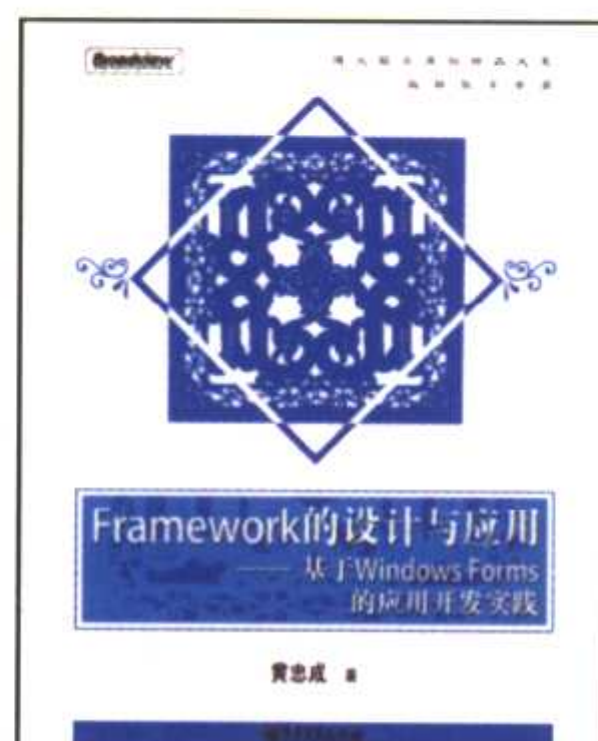
- 国内第一本全面解析SQL Server 2005开发用书
- 数据库专家胡百敬先生（微软MVP）再度引领数据库学习新方向
- 微软推荐SQL Server 2005学习用书



《圣殿祭司的ASP.NET 2.0开发详解——使用C#》

奚江华 著

- 全新ASP.NET 2.0主题
- ASP.NET 2.0账号权限系统彻底解析
- 独家 Atlas Framework AJAX主题
- 台湾微软开发技术推广经理王森先生诚挚推荐



《Framework的设计与应用——基于Windows Forms的应用开发实践》

黄忠成 著

- 剖析框架原理，实践框架应用
- 概念、设计、强化、实践，一应俱全
- 《深入剖析ASP.NET组件设计》作者最新力作





《应用框架的设计与实现——.NET平台》

[美] Xin Chen 著

温 昱 靳向阳 译

- 扎扎实实讲解如何设计应用框架，填补框架设计中文版专著空白
- 揭示框架设计与应用的理论、方法、技术、技巧和最佳实践
- 提供丰富框架案例，助您将设计思想落到实处
- 流畅地将设计模式、.NET高级技术和框架开发思路有机地组织在一起，值得称道

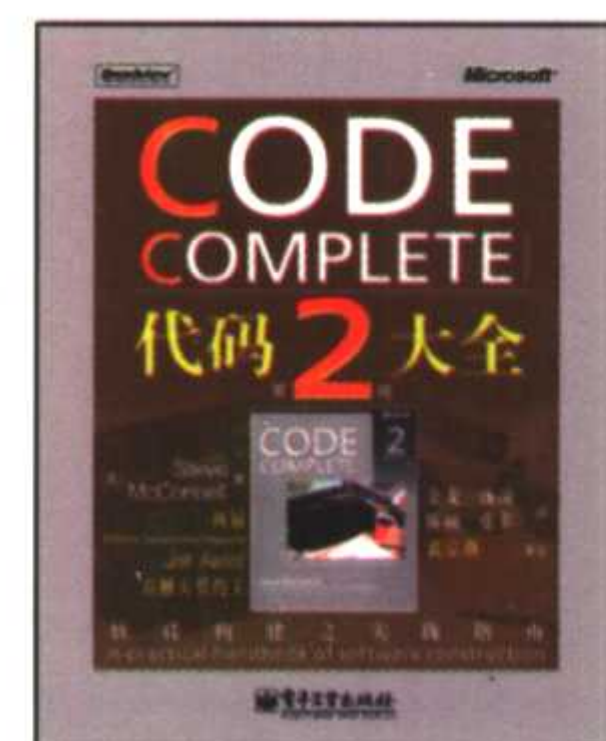


《.NET企业服务框架——应用.NET企业服务开发分布式业务解决方案》

[美] Christian Nagel 著

夏 桅 金雪根 译

- 微软Regional Director、MVP和业界资深技术作者Christian Nagel力作
- 轻松步入服务组件和分布式应用世界
- 知名.NET专家Ingo Rammer鼎力推荐——这世上只有很少几个人能把企业服务解释得足够清楚，而Christian就是其中之一



《代码大全（第2版）》

[美] Steve McConnell 著

金 戈 汤 凌 陈 硕 张 菲 译

- 两届震撼大奖（Jolt Award）得主 Steve McConnell经典著作
- 一年内畅销30000余册
- 十大电脑好书之首（《程序员》杂志、CSDN、第二书店、《中华读书报》2006年评）
- 英文版震撼推出，购买英文版请访问 <http://www.china-pub.com>



技  
术  
凝  
聚  
实  
力

**Broadview**<sup>®</sup>  
WWW.BROADVIEW.COM.CN

专  
业  
创  
新  
出  
版



博文视点——与向上的心合作，共同成长  
专注品质，不断创新，让精益求精成为习惯



## 推荐序

# 架构设计贵在务实

我最早听说“软件架构”这个概念以及 UML 的名字，是在 1999 年的水木清华 BBS 上。当时有一篇文章介绍了软件架构作为一个相对独立的领域的发展情况，顺便提到在此前一年被接纳为 OMG 标准的 UML。该文作者断言，UML 的出现将能“彻底”改变软件开发的工作方式，甚至“若干年之后，不通 UML 者无法染指软件开发”。三年之后，《程序员》杂志专访 Ivar Jacobson 时，UML 已经是尽人皆知。记得 Jacobson 在那次采访中劝告中国的开发者，赶快去学习 RUP。从那时候起，越来越多的人顶上了“软件架构师”的头衔，张口模式闭口架构，一时间好不风光。然而最初的热乎劲过去之后，人们发现，“不通 UML 者无法染指软件开发”的预言似乎落了空，而一些软件架构师们似乎也并不那么神乎其技，很多时候反而不如那些实实在在写代码的人管用。他们所宣传的那些叠床架屋的抽象层，那些复杂精致的模式设计，看上去精美无比，柔性十足，然而实践当中一个出乎意料的小变更，便常常能把这一切打得粉碎。他们乐谈的松耦合，小接口，往往只是说起来好听，实际很难落实，或者代价过高，有的时候，反而是反其道而行之，才更“管用”。

为什么会出现这种情况？我想这里有客观和主观的原因。就客观原因来说，软件开发毕竟还是年轻的行业，各方面还在剧烈发展和变化中。如果把软件技术做一个层次划分的话，软件架构及设计属于上层建筑，而像程序设计语言、技术平台、数据管理技术、网络体系结构等，均在其之下，属于基础。这几年随着互联网的飞速发展，基础尚且在剧烈变化当中，上层建筑自然会摇摇晃晃，甚至赶不上趟。具体来说，当今的软件体系结构设计总体上是基于面向对象思想，而且是强类型语言时代的面向对象思想，而动态语言的出现和流行，实际上很大程度颠覆了传统面向对象思想的一些原则。例如，人们曾经认为封装非常重要，对象成员能够隐藏便应当尽量隐藏，但是 Python 和 Ruby 中 public 是常态，private 反而是变态，实践当中也工作得很好，甚至更好。再例如，几年来人们津津乐道的设计模式，其中有很多在动态语言里毫无必要。而很多在关系数据库时代被视为瑰宝的数据存储与访问模式，放到后关系型数据库里就全无意义了。再诸如应用的 Web 化、RIA、SOA 等基本思想的变迁，都是能引起整个软件技术格局强烈震荡的大事件，所

有这些进行中的剧烈变化，不可能不对软件架构的设计产生影响，从而使得很多关于架构设计的思想迅速过时或者必须调整。如果架构师们不能够充分重视实践，与时俱进，那么就很有可能做出不合时宜的设计。

就主观原因来说，很多软件架构师走入了一个误区，即一旦升级为架构师，就可以脱离具体的代码实践，可以阳春白雪了。事实上，由于下层技术的变化迅速，架构师一旦脱离代码实践，脱离现实应用，很快就会与实实在在的软件开发工作产生距离感，忘却一线开发者需要面对的现实问题，做出一些不切实际的设计决策。这样的设计，或者执行不下去，或者执行下去也代价巨大，该解决的问题没解决，却在无关紧要的问题上大做文章。毫无疑问，这样的设计得不到一线开发者的衷心支持，得不到好的结果。然而，一部分架构师不去检讨自己脱离实践的设计，却搞起本本主义，硬拿书本教条死扣实际。相应地，相当多的开发者也不愿意提高对于软件架构设计的认识和理解。于是两个必要的角色之间产生矛盾。开发者抱怨架构设计华而不实，架构师抱怨开发者不严格按设计行事，进而相互质疑对方角色的必要性。开发者认为架构师没有存在的必要，而架构师则幻想有朝一日自动代码生成器能把这帮不听话的开发者赶出山门。

事实上，开发者和架构师都是软件开发中必不可少的角色，即使在单人开发的项目里，开发者本人也需要经常在这两个角色之间切换。两个角色的相互理解，和谐协作，才能够共同克服现实困难，开发成功的软件。在促进这种和谐的过程中，开发者应当积极学习架构设计的理论并充分实践，而架构师则需要本着务实的态度贴近一线。

因为从事技术媒体工作，我也确实结识了几个优秀的架构设计师，他们身上的共同特点就是务实。这些架构师都具有多年的软件开发经验，对软件本质的理解相当深入，本身就是开发高手。与一般开发高手不同的是，他们充分实践，但不宥于实践，而是积极地学习软件架构的理论，尝试用理论来指导实践。而与整天高谈阔论的理论架构师不同的是，他们掌握了理论之后，一定要亲自落实，用实践来检验。当理论与实践产生矛盾的时候，他们既不会轻易否定理论，更不会教条主义般地削足适履，而是认真分析矛盾产生的原因，研究可能的对策。在反复思考和实践之下，他们敢于做出“离经叛道”的结论，敢于质疑大师偶像的论断，更能够在错综复杂的实际中做出简单、可靠、灵活而便于实现的设计，并且向开发者传达意图，答疑解惑，实现整个团队的思想一致。他们做出的设计，开发者看得懂，做得出，自然会得到衷心的拥护。

温昱是《程序员》杂志的老作者，他本人有丰富的开发经验，为杂志撰写软件架构设计方面的文章也已经有三四年的时间。因为他在上海工作，我与他只见过寥寥几面。然而从与他的交流，以及他的文章来看，温昱也是一位出色的架构设计师。拿到他写的这本书稿之后，我有粗有细地阅读了一遍，感到作者不但理论知识丰富，阅读广泛深入，设计工具运用自如，而且具有务实精神。书中举了大量来自开发一线的实际案例，用这些实际案例当中的设计决策来印证和讲述理论。

当实践与理论发生矛盾的时候，他也是从实践出发，认真地分析原因，找出解决之道。通常我并不喜欢满是 UML 图例的图书，然而温昱这本书读起来却津津有味，不但能够澄清很多观念，学到不少知识，而且阅读过程中有时能进入一种状态，似乎跟着作者一起思考，甚至跟他发生争论。我想这正是因为作者的务实精神。读过此书，再看那些大部头的架构设计经典，理解起来就容易，而且，也许是更重要的，也就具备了一种批判而不是盲从的能力。为此，我乐于向希望学习软件架构设计的开发者和架构师们推荐这本务实之作。

孟岩

2007.3.20

于北京

# 作者序

---

方法如路标。如果地形复杂，我们会迷路，但有了路标，则有利于我们找到前进的方向。好的方法也是如此，它对实践者有启发和指引作用。

我们需要一种有条理的架构设计方法。

这种方法必须有针对性。如何应对需求变更？如何为非功能需求而设计？如何设计架构的不同方面？如何验证架构的可行性？解决这些问题的思路，必须被贯穿到架构设计方法之中，且要显而易见才好。

这种方法必须易于掌握。换句话说，技术要主流；如果都是一线软件人员不熟悉的阳春白雪级的东西，大家掌握起来就比较困难了。

这种方法不能太重。迭代可以为这种方法锦上添花，但不应成为掌握这种方法的障碍；建模也不应滥用，虽然建模不仅有用而且关键……

本书紧紧围绕“软件架构设计”这一主题，立足实践解析了软件架构的概念，阐述了切实可行的软件架构设计方法，提供了可操作性极强的完整的架构设计过程。另外，本书从思维方式的突破、面向对象设计、UML 建模、过程与管理等关键过渡环节，为广大程序员的成长提供了切中肯綮的指导。

理论与实践并重是本书的特点。架构设计要如何开展？架构设计要进行到什么程度？各类需求对架构设计的影响有何不同？关键需求决定架构的具体做法是什么？如何运用“属性—场景—决策”表规划非功能需求？如何运用 OO 原则进行敏捷设计？对这些问题书中都进行了深入阐述，并结合金融、航空、网络管理等行业软件的成功架构设计案例，将理性的思考和宝贵的实践经验奉献给读者。

感谢微软亚洲研究院的刘铁锋、《程序员》杂志的孟岩、BeyondSoft 的夏桅（网名速马）、IBM developerWorks 中国网站的罗景文等朋友的大力支持，他们为本书的策划贡献了真知灼见。感谢我的妻子徐异婕，她对本书的内容、形式、案例均提出了大量宝贵意见和建议。感谢所有为本书提出建议的朋友。

感谢父母长久以来对我的鼓励。感谢岳父、岳母在整个漫长的写作过程中对宝宝无微不至的照顾，使我有更多精力投入在写作上。



由于作者水平有限，本书不足和错误之处在所难免，恳请专家和读者批评指正，欢迎来信（[shanghaiwenyu@163.com](mailto:shanghaiwenyu@163.com)）。本书的支持网站为松耦合空间（[www.ou-he.com](http://www.ou-he.com)），提供与本书内容有关的资源、信息、课件等。

温 昱

2007.2.25

于上海浦东

案例 – 章节对照表

章节	调试系统	网管产品线	PM 系统	银行、超市、多媒体、MIME、在线拍卖、人事管理等案例
1	●		●	
2				
3				
4	●			
5	●			
6		●		
7				
8		●		
9				
10			●	●
11				●
12			●	●
13			●	
14			●	
15			●	
16			●	
17				
18				●
19				●
20				●
21				●
22				●
23				
24				●
25				
26				

## 封面图片说明

---

这本书的封面采用的是一幅精美的剪纸作品，取自袁荃猷先生的《游刃集》（三联书店出版，2002 年）。早年毕业于燕京大学教育系的袁先生，喜爱刻纸，熟谙古琴，性情温婉，多才多艺。她的先生是著名的明清家具收藏家王世襄，也是她的燕京大学校友。才子佳人，琴瑟共鸣。结发 60 载，袁先生跟着王世襄经历了无数的人生风雨，但对文化的热爱和共同追求，让他们在逆境中也能找到生活的乐趣。他们相知相惜，不弃不离，无论条件多么艰苦，都乐于尝试各种形式的艺术创作。

袁先生雅好剪纸，闲暇时光，创作了大量生动有趣的剪纸作品，三联书店特为她出版了这本《游刃集》（海外也发行了繁体版），世襄先生为这本书题签，并作铭曰：“游刃于纸，刃过生文；刃游我游，其乐欣欣。”

封面所采用的这幅剪纸，是一小一大两只美丽的蝴蝶，构图别致典雅，作者看到我们的设计后十分喜欢，说这两只一小一大的蝴蝶正好喻示着他写的书的主题——从程序员到架构师。美编听到作者的反馈后哈哈直乐，因为没想到纯粹从构图的美感上来选择的图片却无意中暗合了作者的写作用心，真是皆大欢喜，写出来以飨读者。

·博·文·视·点·隆·重·推·出·

O'REILLY®



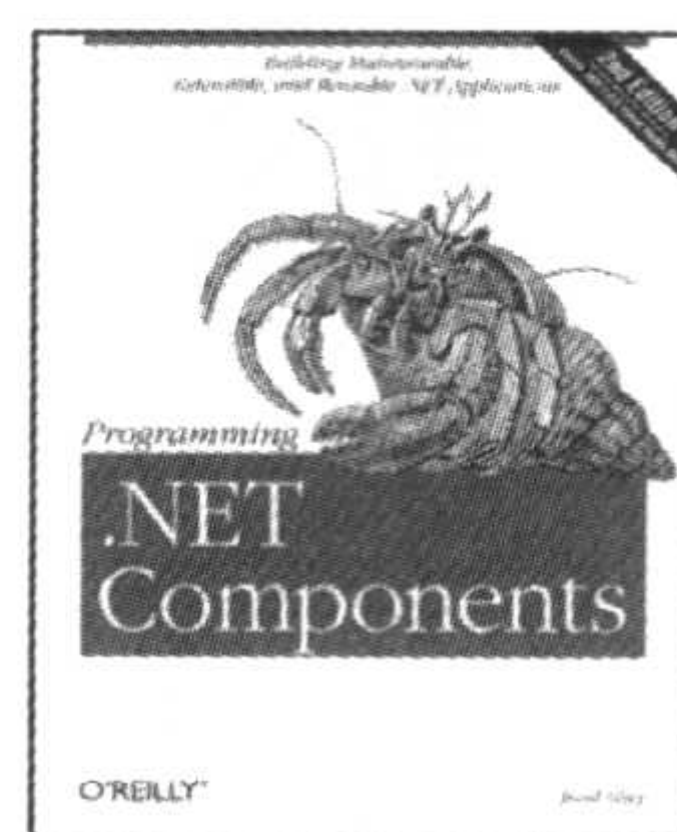
《Programming ASP.NET 中文版 (第3版)》  
Jesse Liberty, Dan Hurwitz 著  
瞿杰 赵立东 张昊 译



《Ajax Hacks 中文版》  
Bruce W. Perry 著



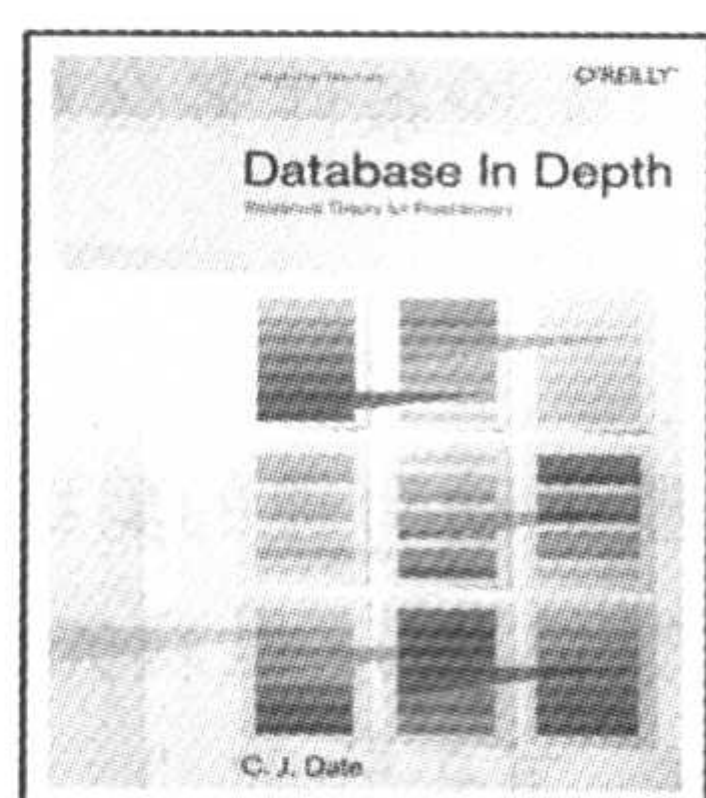
《PHP程序设计, 第2版》  
Rasmus Lerdorf, Kevin Tatroe,  
Peter MacIntyre 著  
陈浩 译



《.NET组件程序设计 (第2版)》  
Juval Löwy 著  
刘如鸿 译



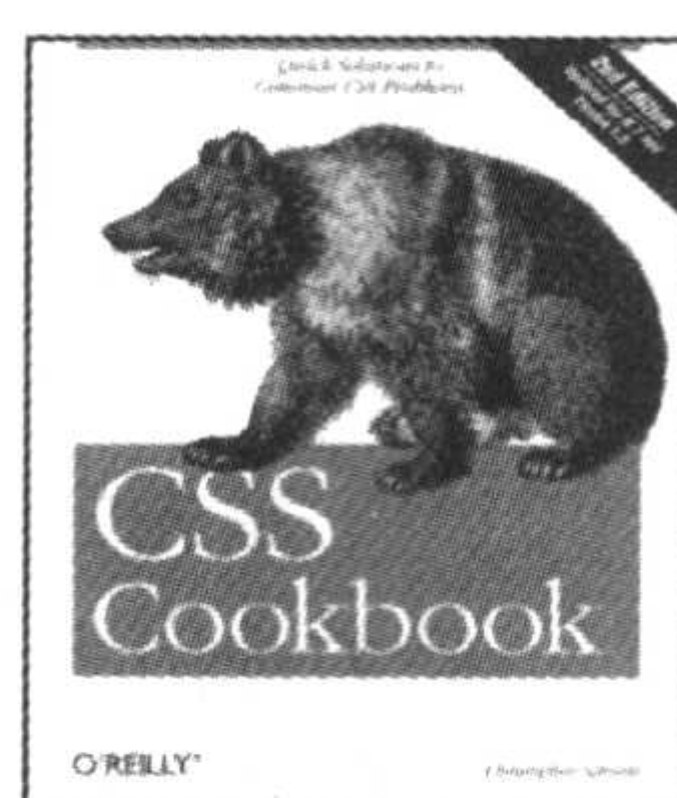
《Programming C#中文版 (第4版)》  
Jesse Liberty 著  
刘基诚 等 译



《Database In Depth中文版》  
C.J. Date 著  
熊建国 译



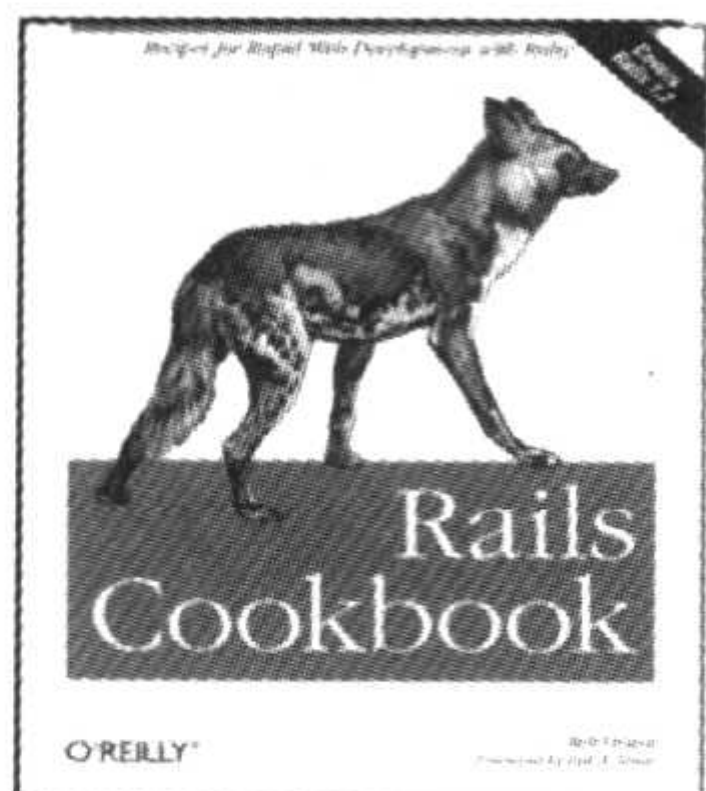
《界面设计精髓》  
Jenifer Tidwell 著



《CSS 实用技巧精粹 (第2版)》  
Christopher Schmitt 著



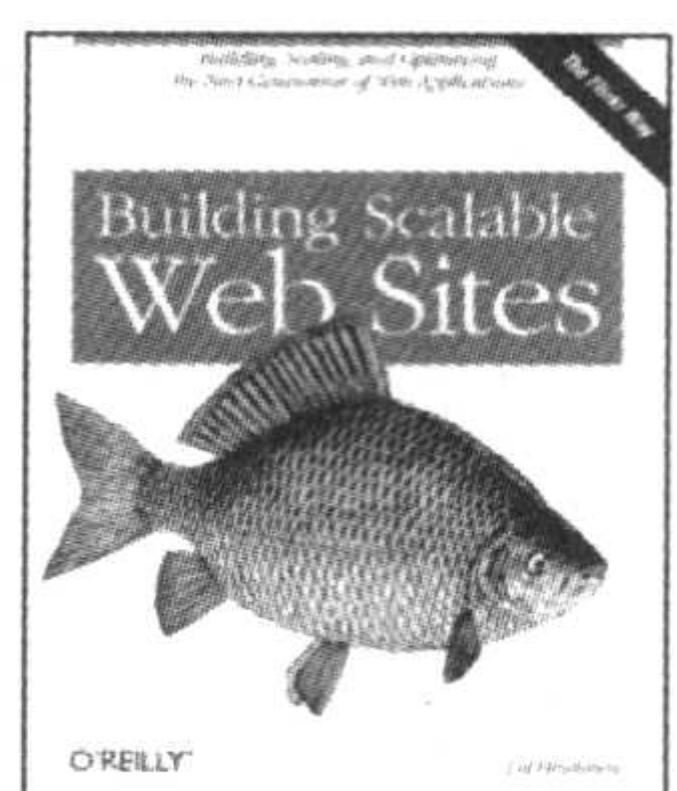
《Ruby In A Nutshell 中文版》  
Yukibiro Matsumoto 著



《Rails Cookbook 中文版》  
Rob Orsini 著



《CSS实战手册》  
David Sawyer McFarland 著



《构建可扩展性Web站点》  
Cal Henderson 著

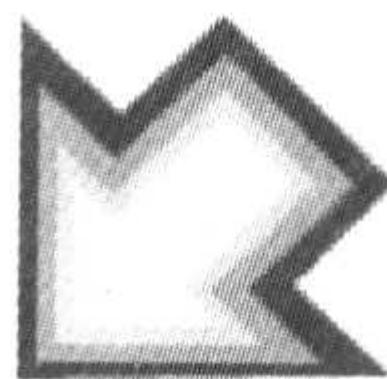


# 软件架构设计

## 有奖读者调查表

欢迎访问以下网站填写调查表，获取更多资源

**<http://bv.csdn.net>**



把您的意见告诉我们，您就有机会获赠博文视点的『**新书一本**』，并参加年终的大抽奖活动！

**Broadview<sup>®</sup>**  
WWW.BROADVIEW.COM.CN

欢迎投稿: [broadvieweditor@gmail.com](mailto:broadvieweditor@gmail.com)

读者信箱: [sheguang@broadview.com.cn](mailto:sheguang@broadview.com.cn)

## 反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396；(010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036

第一部分 软件架构概念与思想篇 .....	1
第 1 章 解析软件架构概念 .....	3
1.1 软件架构概念的分类 .....	3
1.1.1 组成派 .....	4
1.1.2 决策派 .....	5
1.2 软件架构概念大观 .....	5
1.2.1 Booch、Rumbaugh 和 Jacobson 的定义 .....	5
1.2.2 Woods 的观点 .....	6
1.2.3 Garlan 和 Shaw 的定义 .....	6
1.2.4 Perry 和 Wolf 的定义 .....	6
1.2.5 Boehm 的定义 .....	6
1.2.6 IEEE 的定义 .....	6
1.2.7 Bass 的定义 .....	6
1.3 软件架构关注分割与交互 .....	7
1.4 软件架构是一系列有层次性的决策 .....	8
1.5 PM Tool 案例：领会软件架构概念 .....	10
1.5.1 案例故事 .....	10
1.5.2 软件架构概念的体现 .....	12
1.5.3 重要结论 .....	14
1.6 总结与强调 .....	14
第 2 章 子系统、框架与架构 .....	15
2.1 子系统和框架在架构设计中的地位 .....	16
2.1.1 关注点分离之道 .....	16
2.1.2 子系统和框架在架构设计中的地位 .....	17
2.2 子系统与软件架构 .....	19
2.2.1 不同粒度的软件单元 .....	20
2.2.2 子系统也有架构 .....	21
2.2.3 子系统不同，架构不同 .....	21
2.2.4 不同实践者眼中的粒度 .....	23
2.3 框架与软件架构 .....	23
2.3.1 框架的概念 .....	23
2.3.2 架构和框架的区别 .....	24
2.3.3 架构和框架的联系 .....	25



2.3.4 框架也有架构 .....	26
2.4 超越概念：立足实践理解架构 .....	26
2.4.1 理解架构 .....	26
2.4.2 回到实践 .....	28
2.5 专题：框架技术 .....	29
2.5.1 框架 vs. 类库 .....	29
2.5.2 框架的分类 .....	30
2.5.3 框架的开发过程 .....	32
2.5.4 如何实现框架中的扩展点 .....	33
2.6 总结与强调 .....	36
<b>第 3 章 软件架构的作用 .....</b>	<b>37</b>
3.1 充分发挥软件架构的作用 .....	37
3.2 软件架构对新产品开发的作用 .....	38
3.3 软件架构对软件产品线开发的作用 .....	40
3.4 软件架构对软件维护的作用 .....	42
3.5 软件架构重构 .....	42
3.6 总结与强调 .....	43
<b>第二部分 软件架构设计方法与过程篇 .....</b>	<b>45</b>
<b>第 4 章 软件架构视图 .....</b>	<b>47</b>
4.1 呼唤软件架构视图 .....	47
4.1.1 办公室里的争论 .....	48
4.1.2 呼唤软件架构视图 .....	48
4.2 软件架构为谁而设计 .....	49
4.2.1 为用户而设计 .....	49
4.2.2 为客户而设计 .....	50
4.2.3 为开发人员而设计 .....	50
4.2.4 为管理人员而设计 .....	51
4.2.5 总结 .....	51
4.3 引入软件架构视图 .....	52
4.3.1 生活中的“视图”运用 .....	53
4.3.2 什么是软件架构视图 .....	54
4.3.3 多组涉众，多个视图 .....	54
4.4 实践指南：逻辑架构与物理架构 .....	55
4.4.1 逻辑架构 .....	56
4.4.2 物理架构 .....	57
4.4.3 从逻辑架构和物理架构到设计实现 .....	58
4.5 设备调试系统案例：领会逻辑架构和物理架构 .....	59
4.5.1 设备调试系统案例简介 .....	59
4.5.2 逻辑架构设计 .....	59
4.5.3 物理架构设计 .....	61



4.6 总结与强调 .....	62
<b>第 5 章 架构设计的 5 视图法 .....</b>	<b>63</b>
5.1 架构设计的 5 视图法 .....	64
5.2 实践中的 5 视图方法 .....	66
5.3 办公室里的争论：回顾与落实 .....	67
5.4 案例：再谈设备调试系统 .....	67
5.4.1 根据需求决定引入哪些架构视图 .....	68
5.4.2 开发架构设计 .....	68
5.4.3 运行架构设计 .....	69
5.5 总结与强调 .....	71
<b>第 6 章 从概念性架构到实际架构 .....</b>	<b>73</b>
6.1 概念性架构 .....	73
6.2 实际架构 .....	77
6.3 从概念性架构到实际架构 .....	78
6.4 网络管理系统案例：从分层架构开始 .....	78
6.4.1 构思：概念性架构设计 .....	78
6.4.2 深入：实际架构设计 .....	81
6.5 总结与强调 .....	82
<b>第 7 章 如何进行成功的架构设计 .....</b>	<b>83</b>
7.1 何谓成功的软件架构设计 .....	83
7.2 探究成功架构设计的关键要素 .....	84
7.2.1 是否遗漏了至关重要的非功能需求 .....	84
7.2.2 能否驯服数量巨大且频繁变化的需求 .....	86
7.2.3 能否从容设计软件架构的不同方面 .....	86
7.2.4 是否及早验证架构方案并做出了调整 .....	87
7.3 制定软件架构设计策略 .....	87
7.3.1 策略一：全面认识需求 .....	88
7.3.2 策略二：关键需求决定架构 .....	89
7.3.3 策略三：多视图探寻架构 .....	89
7.3.4 策略四：尽早验证架构 .....	90
7.4 总结与强调 .....	90
<b>第 8 章 软件架构要设计到什么程度 .....</b>	<b>93</b>
8.1 软件架构要设计到什么程度 .....	94
8.1.1 分而治之的两种方式 .....	94
8.1.2 架构设计与详细设计 .....	96
8.1.3 软件架构是团队开发的基础 .....	96
8.1.4 架构设计要进行到什么程度 .....	98
8.2 高来高去式架构设计的症状 .....	98

8.2.1	缺失重要架构视图 .....	99
8.2.2	浅尝辄止、不够深入 .....	100
8.2.3	名不副实的分层架构 .....	101
8.3	如何克服高来高去症 .....	101
8.4	网络管理系统案例：如何将架构设计落到实处 .....	102
8.4.1	网管产品线的概念性架构 .....	102
8.4.2	识别每一层中的功能模块 .....	102
8.4.3	明确各层之间的交互接口 .....	103
8.4.4	明确各层之间的交互机制 .....	104
8.4.5	案例小结 .....	105
8.5	总结与强调 .....	105
<b>第 9 章</b>	<b>软件架构设计过程 .....</b>	<b>107</b>
9.1	打造有效的架构设计过程 .....	107
9.1.1	一般的软件过程 .....	107
9.1.2	架构师自己的架构设计过程 .....	109
9.2	软件架构设计过程解析 .....	111
9.2.1	架构设计策略应成为一等公民 .....	111
9.2.2	架构设计过程中的工作产品 .....	112
9.3	总结与强调 .....	114
<b>第 10 章</b>	<b>需求分析 .....</b>	<b>115</b>
10.1	软件需求基础 .....	116
10.1.1	什么是软件需求 .....	116
10.1.2	需求捕获 vs. 需求分析 vs. 系统分析 .....	116
10.1.3	需求捕获及其工作成果 .....	118
10.1.4	需求分析及其工作成果 .....	118
10.1.5	系统分析及其工作成果 .....	119
10.2	需求分析在软件过程中所处的位置 .....	120
10.2.1	概念化阶段所做的工作 .....	120
10.2.2	需求分析所处的位置 .....	122
10.3	架构师必须掌握的需求知识 .....	123
10.3.1	软件需求的类型 .....	123
10.3.2	各类需求对架构设计的影响 .....	127
10.3.3	超市系统案例：领会需求类型的不同影响 .....	129
10.3.4	各类需求的“易变更性”不同 .....	130
10.3.5	质量属性需求与需求折衷 .....	132
10.4	PM Tool 实战：需求分析 .....	135
10.4.1	上游活动：确定项目愿景 .....	135
10.4.2	第 1 步：从业务目标到特性列表 .....	135
10.4.3	第 2 步：从特性列表到用例图 .....	136
10.4.4	第 3 步：从用例图到用例规约 .....	138
10.4.5	需求启发与需求验证 .....	139

10.4.6 最终成果:《软件需求规格说明书》 .....	140
10.5 总结与强调 .....	141
<b>第 11 章 专题:用例技术及应用</b> .....	<b>143</b>
11.1 用例图 vs.用例简述 vs.用例规约 vs.用例实现 .....	143
11.2 储蓄系统案例:需求变化对用例的影响 .....	148
11.3 用例技术应用指南 .....	150
11.4 用例与需求捕获 .....	152
11.5 用例与需求分析 .....	153
11.6 用例与《软件需求规格说明书》 .....	154
11.7 总结与强调 .....	155
<b>第 12 章 领域建模</b> .....	<b>157</b>
12.1 领域模型基础知识 .....	157
12.1.1 什么是领域模型 .....	158
12.1.2 领域模型相关的 UML 图 .....	158
12.2 领域建模在软件过程中所处的位置 .....	159
12.2.1 领域建模的必要性:从需求分析的两个典型困难说起 .....	159
12.2.2 领域建模与需求分析的关系 .....	161
12.2.3 领域建模所处的位置 .....	162
12.3 领域模型对软件架构的重要作用 .....	163
12.3.1 配置管理工具案例:探索复杂问题、固化领域知识 .....	163
12.3.2 人事管理系统案例:决定功能范围、影响可扩展性 .....	165
12.3.3 在线拍卖系统案例:提供交流基础、促进有效沟通 .....	168
12.4 领域模型 vs. 文字说明 .....	170
12.5 PM Tool 实战:建立项目管理的领域模型 .....	171
12.5.1 领域建模实录(1) .....	171
12.5.2 领域建模实录(2) .....	174
12.6 总结与强调 .....	176
<b>第 13 章 确定对软件架构关键的需求</b> .....	<b>177</b>
13.1 虚拟高峰论坛:穷兵黩武还是择战而斗 .....	177
13.1.1 需求是任何促成设计决策的因素 .....	178
13.1.2 很少有开发者能奢侈地拥有一个稳定的需求集 .....	178
13.1.3 关键性的第一步是缩小范围 .....	178
13.1.4 要择战而斗 .....	178
13.1.5 功能、质量和商业需求的某个集合塑造了构架 .....	179
13.2 关键需求决定架构 .....	179
13.2.1 实践中的常见问题 .....	179
13.2.2 关键需求决定架构 .....	181
13.3 确定关键需求在软件过程中所处的位置 .....	182
13.3.1 对架构关键的需求 vs.需求优先级 .....	182

13.3.2 关键需求对后续活动的影响 .....	183
13.4 什么是对软件架构关键的需求 .....	184
13.4.1 关键的功能需求 .....	184
13.4.2 关键的质量属性需求 .....	185
13.4.3 关键的商业需求 .....	186
13.5 如何确定对软件架构关键的需求 .....	187
13.5.1 全面整理需求 .....	188
13.5.2 分析约束性需求 .....	188
13.5.3 确定关键功能需求 .....	189
13.5.4 确定关键质量属性需求 .....	190
13.6 PM Tool 实战：确定关键需求 .....	190
13.7 总结与强调 .....	191
<b>第 14 章 概念性架构设计 .....</b>	<b>193</b>
14.1 概念性架构设计的步骤 .....	194
14.2 鲁棒性分析 .....	195
14.2.1 分析和设计之间的鸿沟 .....	195
14.2.2 鲁棒图简介 .....	196
14.2.3 从用例到鲁棒图 .....	197
14.3 运用架构模式 .....	198
14.3.1 架构模式简介 .....	198
14.3.2 架构模式的经典分类 .....	199
14.3.3 架构模式的现代分类 .....	200
14.3.4 分层 .....	201
14.3.5 MVC .....	201
14.3.6 微内核 .....	202
14.3.7 基于元模型的架构 .....	203
14.3.8 管道—过滤器 .....	204
14.4 PM Tool 实战：概念性架构设计 .....	204
14.4.1 进行鲁棒性分析 .....	204
14.4.2 引入架构模式 .....	206
14.4.3 质量属性分析 .....	207
14.4.4 设计结果 .....	207
14.5 总结与强调 .....	208
<b>第 15 章 质量属性分析 .....</b>	<b>209</b>
15.1 质量属性需求基础 .....	210
15.2 质量属性分析的位置 .....	211
15.3 利用“属性—场景—决策”表设计架构决策 .....	211
15.3.1 概述 .....	211
15.3.2 “属性—场景—决策”表方法 .....	212
15.3.3 题外话：《需求文档》如何定义质量属性需求 .....	214
15.4 PM Tool 实战：可扩展性设计 .....	214



15.5 总结与强调 .....	215
<b>第 16 章 细化架构设计 .....</b>	<b>217</b>
16.1 架构细化在软件过程中所处的位置 .....	218
16.1.1 我们走到哪了 .....	218
16.1.2 运用基于 5 视图方法进行架构细化 .....	219
16.2 设计逻辑架构 .....	220
16.2.1 概述 .....	220
16.2.2 识别通用机制 .....	220
16.3 设计开发架构 .....	223
16.3.1 概述 .....	223
16.3.2 分层和分区 .....	223
16.4 设计数据架构 .....	226
16.4.1 概述 .....	226
16.4.2 如何将 OO 模型映射为数据模型 .....	227
16.5 设计运行架构 .....	229
16.5.1 概述 .....	229
16.5.2 运用主动类规划并发 .....	230
16.5.3 应用协议的设计 .....	234
16.6 设计物理架构 .....	234
16.6.1 概述 .....	234
16.7 注意满足所有约束性软件需求 .....	235
16.8 PM Tool 实战：细化架构设计 .....	236
16.9 总结与强调 .....	239
<b>第 17 章 实现并验证软件架构 .....</b>	<b>241</b>
17.1 基础知识 .....	242
17.1.1 原型技术及分类 .....	242
17.1.2 验证架构的两种方法 .....	245
17.2 实现并验证软件架构的具体做法 .....	245
17.3 总结与强调 .....	247
<b>第三部分 程序员成长篇 .....</b>	<b>249</b>
<b>第 18 章 MIME 编码类案例：从面向过程到面向对象 .....</b>	<b>251</b>
18.1 设计目标 .....	251
18.2 MIME 编码基础知识 .....	252
18.3 MIME 编码类的设计过程 .....	252
18.3.1 面向过程的设计方案 .....	252
18.3.2 转向面向对象设计 .....	254
18.3.3 面向对象设计方案的确定 .....	257
18.3.4 Template Method 和 Strategy 模式的对比 .....	260

第 19 章 突破 OOP 思维：继承在 OOD 中的应用.....	261
19.1 从一则禅师语录说起.....	261
19.1.1 见继承是继承——程序员境界.....	262
19.1.2 见继承不是继承——成长境界.....	262
19.1.3 见继承只是继承——设计师境界.....	262
19.2 从 OOD 层面认识继承.....	262
19.3 针对接口编程——隔离变化.....	263
19.3.1 相关理论.....	263
19.3.2 针对接口编程举例——用于架构设计.....	263
19.3.3 针对接口编程举例——用于类设计.....	265
19.4 混入类——更好的重用性.....	266
19.4.1 相关理论.....	266
19.4.2 混入类举例.....	266
19.5 基于角色的设计——使用角色组装协作.....	267
19.5.1 相关理论.....	267
19.5.2 基于角色的设计举例.....	268
第 20 章 细微见真章：耦合其实并不空洞.....	269
20.1 顺序耦合性简介.....	269
20.2 案例研究：顺序耦合性 Bug 一例.....	269
20.2.1 项目简介.....	270
20.2.2 新的需求.....	270
20.2.3 发现顺序耦合性 Bug.....	271
20.2.4 跟踪调试.....	271
20.2.5 分析原因.....	273
20.2.6 解决策略.....	273
20.2.7 运用重构的“Extract Method”成例.....	273
20.2.8 运用重构的“Hide Method”成例.....	274
20.2.9 运用重构的“Introduce Parameter Object”成例.....	274
20.2.10 其他改进.....	274
第 21 章 敏捷设计：从理论到实践.....	277
21.1 换个角度考察依赖.....	278
21.1.1 依赖的概念.....	278
21.1.2 从会不会造成“实际危害”的角度考察依赖.....	278
21.2 良性依赖原则.....	278
21.2.1 依赖是不可避免的.....	278
21.2.2 重要的是如何务实地应付变化.....	279
21.3 案例：需求改变引起良性依赖变成恶性依赖.....	279
21.4 案例：隔离第三方 SDK 可能造成的冲击.....	281
21.5 案例：对具体类的良性依赖.....	283
21.6 总结：如何处理好依赖关系.....	285

<b>第 22 章 基于角色的设计：从理论到实践</b>	<b>287</b>
22.1 基于角色的设计理论	288
22.2 基于角色的设计与团队开发	288
22.3 基于角色的设计实践	289
22.4 基于角色的设计案例	291
22.4.1 项目简介	291
22.4.2 通过基于角色的设计组织子系统之间的协作	291
22.4.3 通过基于角色的设计组织同一子系统内不同模块之间的协作	292
22.5 基于角色的设计与面向对象分析	293
<b>第 23 章 超越设计模式：理解和运用更多模式</b>	<b>295</b>
23.1 关于模式的两个问题	295
23.2 模式的正交分类法	296
23.2.1 正交思维	296
23.2.2 正交思维用于模式分类	297
23.3 专攻性能：性能模式简介	299
23.4 模型驱动开发的方方面面：MDD 模式简介	301
23.5 总结：拥抱模式	302
<b>第 24 章 如此轻松：立足图论学 UML</b>	<b>303</b>
24.1 管窥 UML 中的 OO 思想	304
24.1.1 一道笔试题的故事	304
24.1.2 UML 背后的思想	305
24.2 图的定义与 UML 应用	306
24.2.1 图的定义	306
24.2.2 图的定义的 UML 应用——UML 的图论观点	307
24.2.3 图的定义的 UML 应用——关联类语法的理解	308
24.2.4 图的定义的 UML 应用——说说序列图	309
24.3 有向边与 UML 应用	310
24.3.1 有向边	310
24.3.2 有向边的 UML 应用——依赖关系	310
24.3.3 有向边的 UML 应用——泛化、实现和关联的依赖思想	312
24.3.4 有向边的 UML 应用——一个例子	312
24.4 着色顶点与 UML 应用	313
24.4.1 着色顶点	313
24.4.2 着色顶点的 UML 应用——通过颜色为图元分类	314
24.4.3 着色顶点的 UML 应用——UML 彩色建模方法介绍	315
24.5 着色边与 UML 应用	317
24.6 图的同构与 UML 应用	317
24.6.1 图的同构	317
24.6.2 图的同构的 UML 应用——UML 风格	318

第 25 章 理解软件过程：解析 RUP 核心概念 .....	321
25.1 架构师必须了解软件过程 .....	321
25.1.1 架构师的工作职责 .....	321
25.1.2 架构师必须了解软件过程 .....	322
25.2 RUP 实践中的常见问题 .....	322
25.3 RUP 核心概念解析 .....	323
25.3.1 一图胜千言 .....	323
25.3.2 角色执行活动，活动生产工件 .....	323
25.3.3 阶段和迭代：提供不同级别的决策时机 .....	324
25.3.4 配置和变更管理支持迭代式的基于基线的开发 .....	326
25.3.5 发布是什么，发布不是什么 .....	327
第 26 章 海阔凭鱼跃：通盘理解软件工程 .....	329
26.1 什么是软件工程概念模型 .....	329
26.2 一个精简的软件工程概念模型 .....	329
26.3 一个细化的软件工程概念模型 .....	330
26.3.1 模型概述 .....	331
26.3.2 方法论 .....	331
26.3.3 过程 .....	331
26.3.4 目标 .....	332
26.3.5 项目 .....	332
26.3.6 其他 .....	333
26.4 软件工程概念模型的具体应用 .....	333
26.4.1 搞清楚 Agile 是过程还是方法论 .....	333
26.4.2 为 CMM 定位 .....	334
26.4.3 理解 RUP 定制 .....	335
26.5 总结：软件工程概念模型的启示 .....	335
26.5.1 软件工程，一门实践的科学 .....	335
26.5.2 软件过程，合适的才是最好的 .....	336
26.5.3 对个人的启示 .....	336
26.5.4 呼唤高层次人才 .....	336
参考文献 .....	337



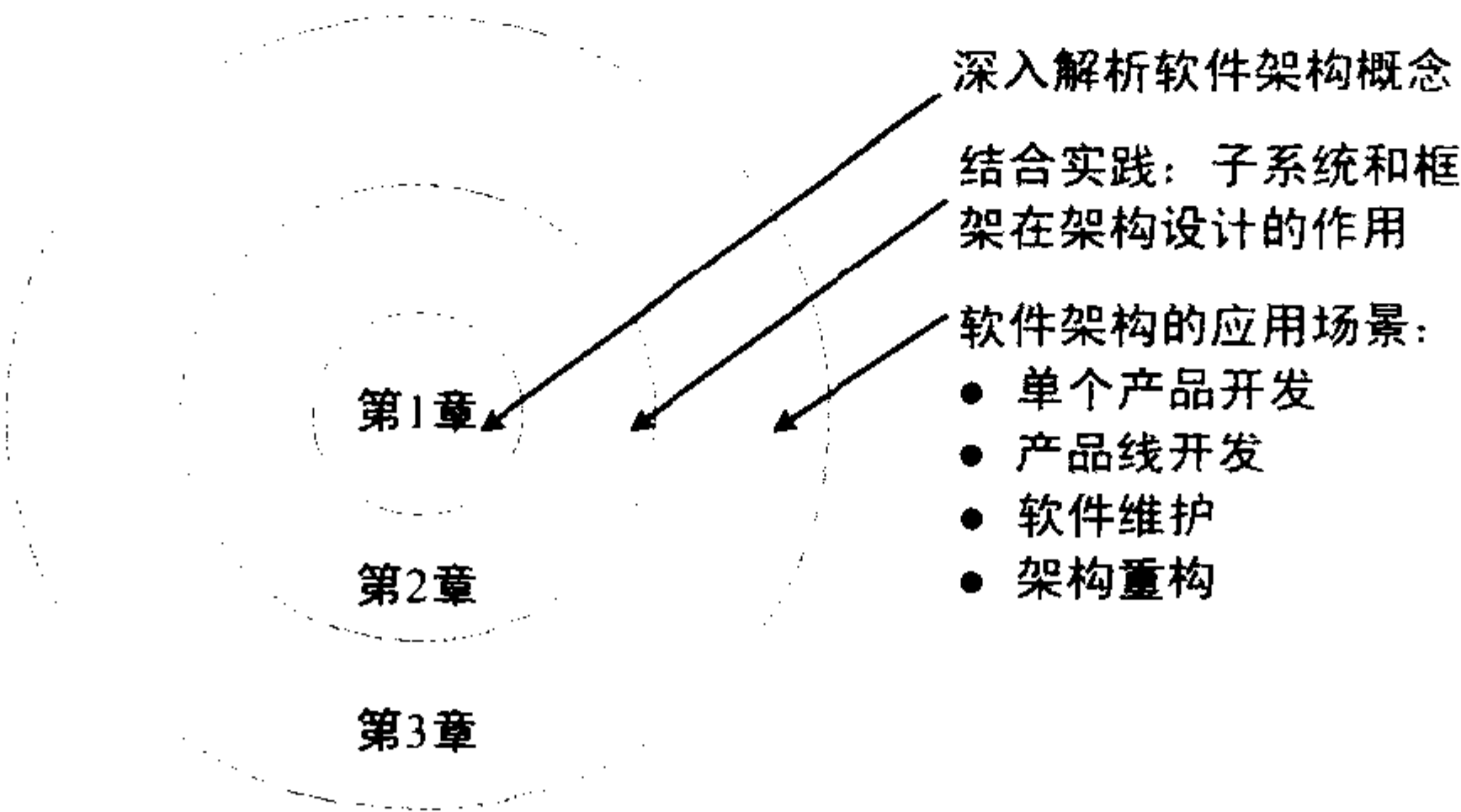
# 第一部分

## 软件架构概念与思想篇

作为本书的第一部分，前 3 章要讨论的主题分别是：

- 解析软件架构（Architecture）概念
- 子系统、框架（Framework）与架构
- 软件架构的作用

下图揭示了这 3 章的关系：理解架构概念是基础；掌握子系统、框架和架构的关系是实践的要求；而软件架构最终用于实践，在单个产品开发、产品线开发、软件维护、软件升级等不同情况下都有重要的作用。





# 第 1 章 解析软件架构概念

---

不积跬步，无以至千里。——《荀子·劝学篇》

什么是架构？如果你问五个不同的人，可能会得到五种不同的答案。

——Ivar Jacobson, 《AOSD 中文版》

很多人都试图给“架构”下定义，而这些定义本身却很难统一。

——Martin Fowler, 《企业应用架构模式》

不积跬步，无以至千里。作为本书的第 1 章，我们首先讨论软件架构的概念。为了给读者带来实感，我们将结合流行的 MVC 架构以及一个名为 PM Tool 的项目管理系统的案例故事进行讲解。

值得说明的是，人们对“Architecture”有着不同的中文叫法，比如架构、构架和体系结构等。本书将一以贯之地采用“架构”的叫法；当然，当引用原文或提及书名时将保留原来的叫法。

## 1.1 软件架构概念的分类

---

一个词（比如“电脑”），可能并不代表一件单独的东西，而是代表了一类事物。这个一般性的表述就是我们通常所说的“概念”。

也许读者期待一个干净利落的软件架构概念，但这有点儿难。对此，Martin Fowler 给出的评价是，“软件业的人乐于做这样的事——找一些词汇，并将它们引申到大量微妙而又相互矛盾的含义中。一个最大的受害者就是‘架构’这个词。……很多人都试图给‘架构’下定义，而这些定义本身却很难统一。”

的确如此。由于软件架构的概念有太多的版本，这已经对软件架构师的工作，乃至整个软件开发活动造成了不少麻烦。

- **造成交流上的误解。**此君说的软件架构，和彼君理解的软件架构未必是一回事；
- **造成实践上的障碍。**软件架构设计领域和其他任何复杂领域一样，即使对基本概念和思想有了清晰的认识，依然未必能够保证实践畅通无阻。更何况是在架构概念本身还处于“微妙的相互矛盾”之中的情况下呢？
- **造成分工合作中的不一致。**架构设计究竟要覆盖哪些范围？架构设计应进行到什么程度？对这两个问题认识上的不一致当然也和架构概念混乱有关。一方面，开发者对架构的明确程度期望甚高；另一方面，架构师的架构设计方案却是高来高去；这就造成了大量设计空白，最终往往靠临时拍脑袋或临时商量来决定，于是软件质量下降、协作开发混乱就在所难免了；
- **造成其他概念乘虚而入。**我们的大脑中“驻扎”了很多概念，“软件架构”一词如此流行，以至于每个软件人员的大脑中都有它的一席之地。但由于软件架构的概念本身没有一个具有压倒性优势的定義充当“旗手”，致使许多人大脑中贴着“软件架构”标签的位置已经被其他概念占据了——典型的片面认识包括架构就是结构、架构就是基础设施（infrastructure）、架构就是框架（Framework），等等。

本书认为，鉴于软件架构概念的混乱，应该采取分类的办法。这是因为，通过分类可以在包容细节差异的基础上明确共性，达到“概念总体上的清晰”。笔者的经验证明，这种务实的做法是有利于实践的。

下面，我们将软件架构概念分为两大流派：组成派和决策派。

### 1.1.1 组成派

Mary Shaw 在《软件体系结构：一门初露端倪学科的展望》中，为“软件架构”给出了非常简明的定义：

软件系统的架构将系统描述为计算组件及组件之间的交互（The architecture of a software system defines that system in terms of computational components and interactions among those components.）。

必须说明，上述定义中的“组件”是广泛意义上的元素之意，并不是指和 CORBA、DCOM、EJB 等相关的专有的组件概念。“计算组件”也是泛指，其实计算组件可以进一步细分为处理组件、数据组件、连接组件等。总之，“组件”可以指子系统、框架（Framework）、模块、类等不同粒度的软件单元，它们可以担负不同的计算职责。

上述定义是“组成派”软件架构概念的典型代表，有如下两个显著特点：

- （1）关注架构实践中的客体——软件，以软件本身为描述对象；
- （2）分析了软件的组成，即软件由承担不同计算任务的组件组成，这些组件通过相互交互完成更高层次的计算。



### 1.1.2 决策派

下面来看看 RUP (Rational Unified Process, Rational 统一过程) 中的软件架构定义。

软件架构包含了关于以下问题的重要决策:

- 软件系统的组织;
- 选择组成系统的结构元素和它们之间的接口, 以及当这些元素相互协作时所体现的行为;
- 如何组合这些元素, 使它们逐渐合成为更大的子系统;
- 用于指导这个系统组织的架构风格: 这些元素以及它们的接口、协作和组合。

软件架构并不仅仅注重软件本身的结构和行为, 还注重其他特性: 使用、功能性、性能、弹性、重用、可理解性、经济和技术的限制及权衡, 以及美学等。

上述定义看似冗长, 但其核心思想非常明确: 软件架构是在一些重要方面所作出的决策的集合。

该定义是“决策派”软件架构概念的典型代表, 有如下两个显著特点:

- (1) 关注架构实践中的主体——人, 以人的决策为描述对象;
- (2) 归纳了架构决策的类型, 指出架构决策不仅包括关于软件系统的组织、元素、子系统和架构风格等几类决策, 还包括关于众多非功能需求的决策。

## 1.2 软件架构概念大观

如前所述, 将软件架构概念分类的好处是: 包容细节差异、明确本质共性、促成概念总体上的清晰。下面我们再列举几个著名的软件架构定义, 以期达到下列目的:

- (1) 体会和证明众多软件架构概念都是围绕“组成”和“决策”两个视角展开的;
- (2) 开阔视野, 说不定和你合作的同事所接受的软件架构概念就是其中的一种。

具体而言, 下面的定义 1 和定义 2 属于架构概念的“决策派”, 而定义 3、4、5、6、7 属于架构概念的“组成派”。值得说明的是, 定义 7 是来自 SEI 的 Bass 等人的相对比较新的定义, 它将架构的多视图“本性”体现到了架构的定义当中, 本书认为这种做法非常值得肯定。在第 4 章中, 我们将专门讨论软件架构视图这一主题。

### 1.2.1 Booch、Rumbaugh 和 Jacobson 的定义

架构是一系列重要决策的集合, 这些决策与以下内容有关: 软件的组织, 构成系统的结构元

素及其接口的选择，这些元素在相互协作中明确表现出的行为，这些结构元素和行为元素进一步组合所构成的更大规模的子系统，以及指导这一组织——包括这些元素及其接口、它们的协作和它们的组合——架构风格。

### 1.2.2 Woods 的观点

Eoin Woods 是这样认为的：软件架构是一系列设计决策，如果作了不正确的决策，你的项目可能最终会被取消（Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.）。

### 1.2.3 Garlan 和 Shaw 的定义

Garlan 和 Shaw 认为：架构包括组件（Component）、连接件（Connector）和约束（Constrain）三大要素。组件可以是一组代码（例如程序模块），也可以是独立的程序（例如数据库服务器）。连接件可以是过程调用、管道和消息等，用于表示组件之间的相互关系。“约束”一般为组件连接时的条件。

### 1.2.4 Perry 和 Wolf 的定义

Perry 和 Wolf 提出：软件架构是一组具有特定形式的架构元素，这些元素分为三类：负责完成数据加工的处理元素（Processing Elements）、作为被加工信息的数据元素（Data Elements）及用于把架构的不同部分组合在一起的连接元素（Connecting Elements）。

### 1.2.5 Boehm 的定义

Barry Boehm 和他的学生提出：软件架构包括系统组件、连接件和约束的集合，反应不同涉众需求的集合，以及原理（Rationale）的集合。其中的原理，用于说明由组件、连接件和约束所定义的系统在实现时，是如何满足不同涉众需求的。

### 1.2.6 IEEE 的定义

IEEE 610.12-1990 软件工程标准词汇中是这样定义架构的：架构是以组件、组件之间的关系、组件与环境之间的关系为内容的某一系统的基本组织结构，以及指导上述内容设计与演化的原理（Principle）。

### 1.2.7 Bass 的定义

SEI（Software Engineering Institute, SEI, 美国卡内基梅隆大学软件研究所）的 Bass 等人给架构的定义是：某个软件或计算机系统的软件架构是该系统的一个或多个结构，每个结构均由软



件元素、这些元素的外部可见属性、这些元素之间的关系组成（The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.）。

### 1.3 软件架构关注分割与交互

本节结合流行的 MVC 架构，说明软件架构是如何将系统分为不同部分，以及各部分之间如何交互的。

“软件系统的架构将系统描述为计算组件及组件之间的交互”，Shaw 的这个定义从“软件组成”角度解析了软件架构的要素：组件及组件之间的交互。通过 UML 类图，我们可以将这个关系表达得更加清晰，参见图 1-1。如图所示，组件和组件之间有交互关系，图中的“交互”采用了关联类的语法。

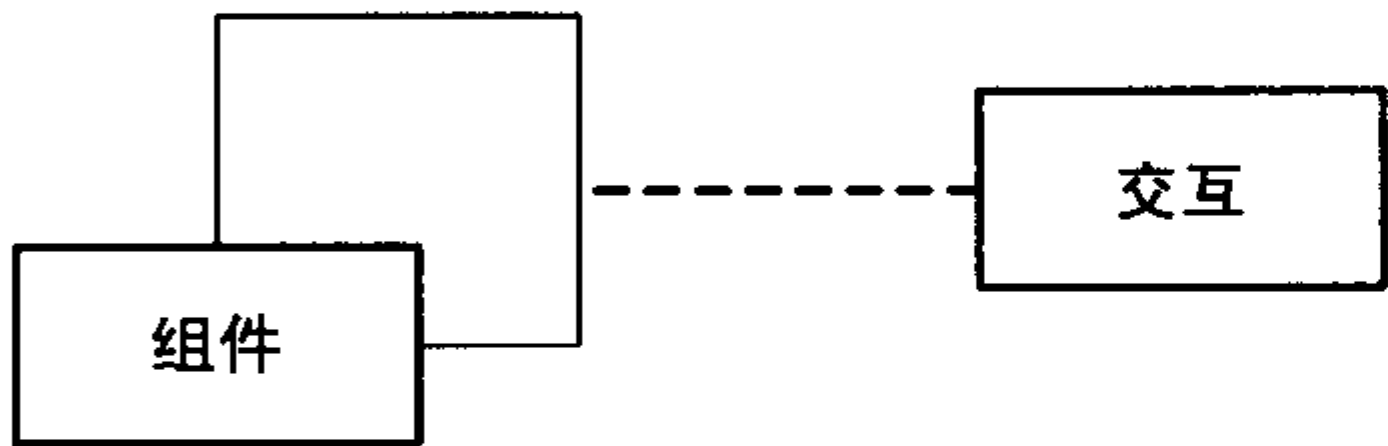


图 1-1 软件架构的要素：组件及组件之间的交互

Shaw 的架构定义高度抽象地将软件架构概括为“组件+交互”，下面以大家熟悉的 MVC 架构为例进行说明（如图 1-2 所示）。

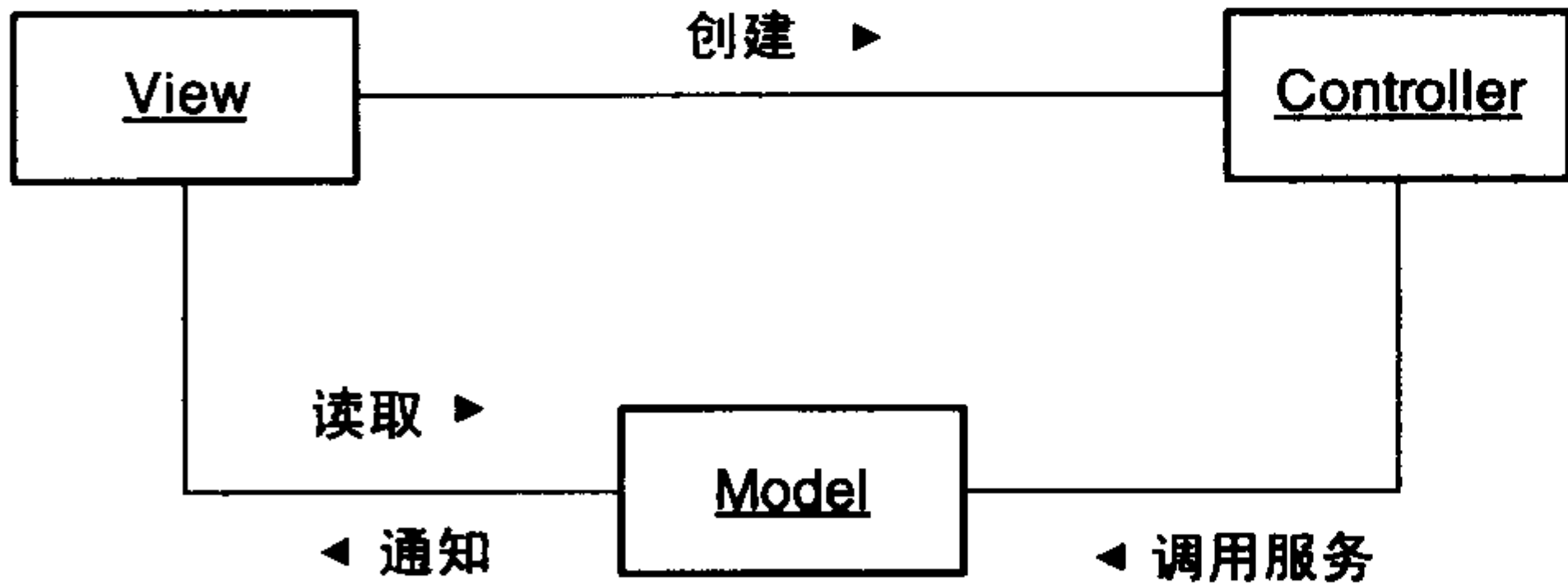


图 1-2 MVC 架构作为“组件+交互”的例子

采用 MVC 架构的软件包含了这样 3 种组件：Model、View、Controller。

同时，为了达到一定的目的，这 3 种组件必须通过交互来协作，比如：View 创建 Controller 后，Controller 根据用户交互调用 Model 的相应服务，而 Model 会将自身的改变通知 View，View 则会读取 Model 的信息以更新自身。

通过此例可以看出，“组件+交互”可以将 MVC 等“具体架构设计决策”高屋建瓴地抽象地表达出来。

# 1.4 软件架构是一系列有层次性的决策

软件架构属于设计范畴，但并不是所有设计都属于软件架构设计之列。

正如前面软件架构的“决策派”概念所揭示的，软件架构可以视为一系列重要决策的集合。不仅如此，架构决策还是分层次依次展开的。

首先，伴随着对软件系统的依次分解，软件架构师应当不断作出决策，例如需要划分成哪些模块，每个模块的职责为何，每个模块的接口如何定义，模块间采用何种交互机制，如何满足约束和质量属性的需求，如何适应可能发生的变化等。

以一个硬件设备调试系统为例。软件架构师通过理解需求，对该设备调试系统应完成的主要目标有了全局的把握：作为设备调试系统，其主要功能是实时显示设备状态，以及支持用户发送调试命令；另外，由于是为硬件产品配套的软件系统，所以它必须容易被测试，否则是硬件故障还是软件故障将很难区分；再就是必须具有很高的性能，具体性能指标为“每秒钟能够刷新 5 次设备状态的显示，并同时支持一个完整命令字的发送”。如图 1-3 所示。

设备调试系统	协作者
职责	被调试设备 数据采集器
设备状态的实时显示 支持用户发送调试命令 易测试性 高性能	

图 1-3 CRC 卡：设备调试系统应完成的主要目标

之后，软件架构师必须规划整个系统的具体组成。通常，对于一个独立的软件系统而言，它常常被划分为不同的子系统或分系统，每个部分承担相对独立的功能，各部分之间通过特定的交互机制进行协作。而此例中的设备调试系统则不同，它有两个相对独立的应用组成：一个桌面应用和一个嵌入式应用。那么，它们如何通讯呢？最终决定，将它们通过串口连接，采用 RS232 协议进行通讯。再接下来，架构师必须决定这两个应用分别担负哪些职责。最终的设计决策是（如图 1-4 中的 CRC 卡所示）：桌面应用部分负责提供模拟控制台和状态显示；而嵌入式应用部分负责设备的控制和状态数据的读取。



桌面应用	协作者 嵌入式应用	嵌入式应用	协作者 桌面应用 被调试设备 数据采集器
<b>职责</b> 负责设备状态的显示 提供模拟控制台供用户发送调试命令 通过串口和嵌入式应用通讯		<b>职责</b> 负责对调试设备的具体控制 高频度地从数据采集器读取设备状态数据 通过串口和桌面应用通讯	

图 1-4 CRC 卡：设备调试系统的组成部分

设备调试系统的桌面应用部分是一个复杂的应用程序，还应当由软件架构师来负责该应用的架构设计。如图 1-5 所示，通讯部分被分离出来作为通讯层，它负责在 RS232 协议之上实现一套专用的“应用协议”：当应用层发送来包含调试指令的协议包时，它会按 RS232 协议将之传递给嵌入部分；当嵌入部分发送来原始数据时，它将之解释成应用协议包发送给应用层。而应用层负责设备状态的显示，提供模拟控制台供用户发送调试命令，并使用通讯层和嵌入部分进行交互。

桌面应用::应用层	协作者 通讯层	桌面应用::通讯层	协作者 嵌入式应用
<b>职责</b> 负责设备状态的显示 提供模拟控制台供用户发送调试命令 使用通讯层和嵌入部分进行交互		<b>职责</b> 负责在RS232协议之上实现一套专用的应用协议” 当应用层发送来包含调试指令的协议包，负责按RS232协议将之传递给嵌入部分 当嵌入部分发送来原始数据，将之解释成应用协议包发送给应用层	

图 1-5 CRC 卡：桌面应用进一步分解

通过上面的这个案例，我们强烈地感觉到：软件架构师需要作出的一系列重要设计决策，是伴随着对软件系统的层层分解依次展开的。例如，不将整个设备调试系统分为桌面部分和嵌入式部分，也就无从将桌面部分进一步分离成应用层和通讯层。

如前所述，其实设计决策不仅仅局限在职责划分上。例如，应用层应当如何设计才能保证很高的用户响应速度呢？通讯层应当如何设计以保证高性能地接受串口数据而不造成数据丢失呢？这些也都是架构师需要考虑的，在此不再赘述（可参考第 4 章和第 5 章的案例分析部分）。

其次，“架构决策是分层次依次展开的”还表现在：决策制定的顺序往往是先制定技术无关的决策，后制定技术相关的决策，后者在前者的指导下进行。对于这一点，本书将在软件架构设计过程的讲解中讨论（参见本书第二部分中软件架构设计方法与过程篇）。

1.5 PM Tool 案例：领会软件架构概念

为了有助于领会软件架构的概念，我们引入一个名为“PM Tool”的案例。PM Tool 是一个项目管理系统，提供项目计划、任务管理和资源管理等功能。本节希望结合案例来讨论软件架构，为读者理解软件架构的概念带来实感。

1.5.1 案例故事

PM Tool 有一项名为“查看甘特图”的需求，用户要求“能够以甘特图方式查看任务的起始时间、结束时间、任务承担者等信息”。经过分析我们不难发现，PM Tool 至少应提供两种查看任务计划的方式：一种是以表格的方式将任务名称、开始时间等信息列出，另一种是采用甘特图。如图 1-6 所示。

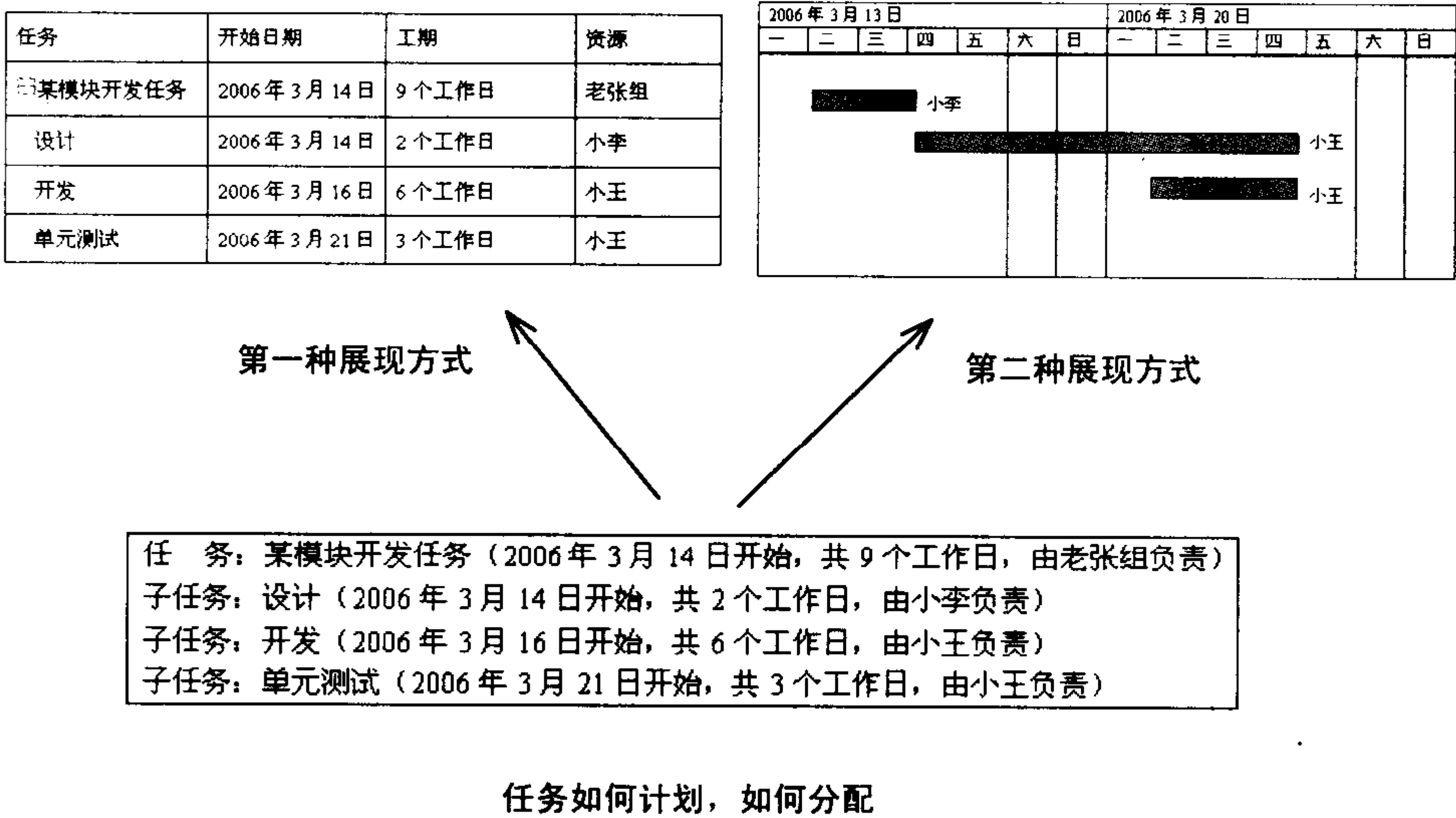


图 1-6 PM Tool 至少需要提供两种查看任务计划的方式

任务是如何计划的？又具体分配给哪些项目成员？这些信息和 PM Tool 采用何种方式来展现应当是没有关系的。根据此分析，我们立即想到采用 MVC 架构，将业务逻辑和展现逻辑分开，如图 1-7 所示。



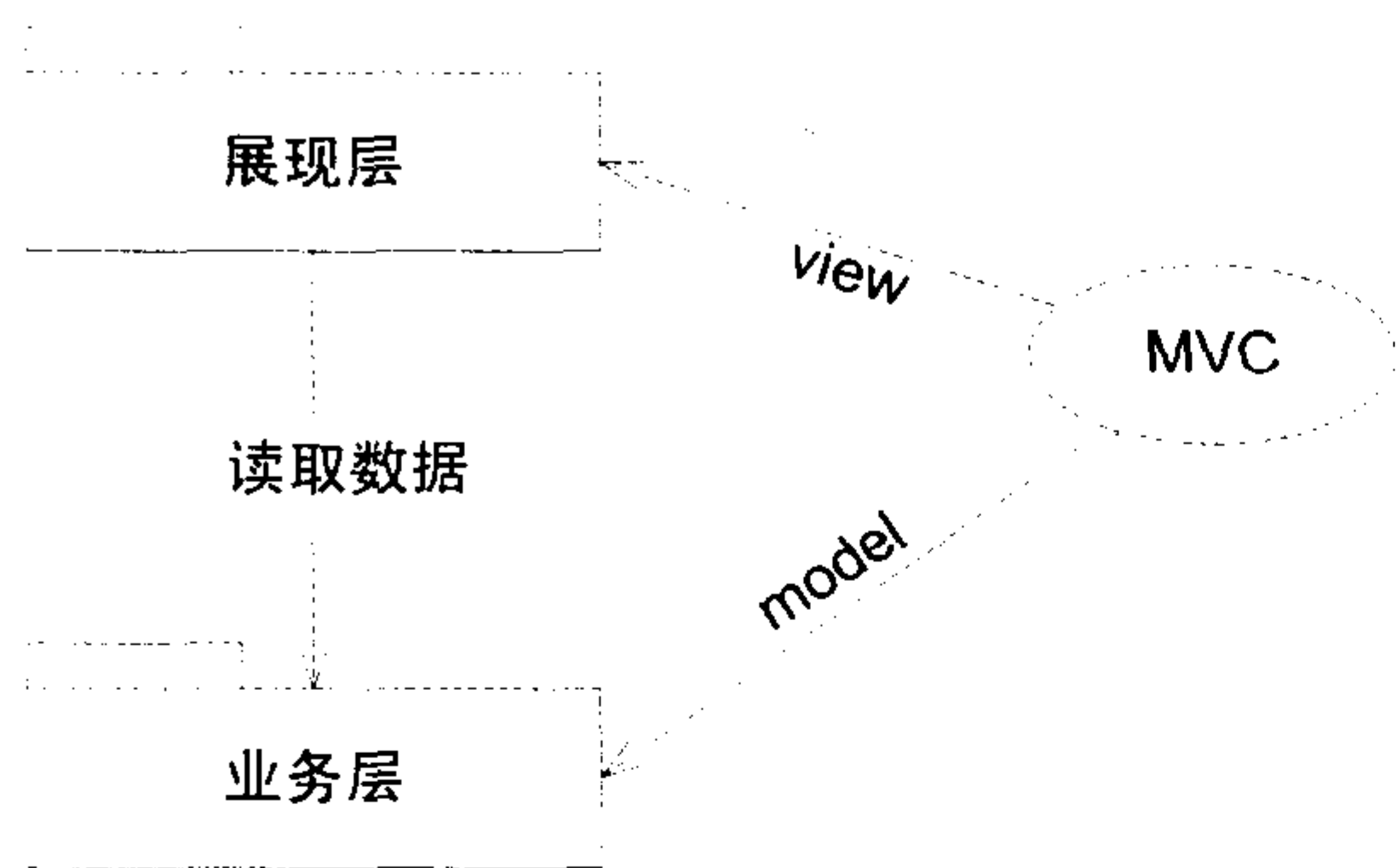


图 1-7 和具体技术无关的架构方案

上面的架构设计，还处于“和具体技术无关”的层面。我们必须考虑更多的实际开发中要涉及到的技术问题，从而不断细化架构方案，这样才能为开发人员提供更多的指导和限制，也才能真正降低后续开发中的重大技术风险。

对于 PM Tool 要显示甘特图而言，“甘特图绘制包”是自行开发的，还是采用的第三方 SDK，就是一个很重要的技术问题。考虑如下：

一方面，用户根本不关心“甘特图绘制包”的问题，他们只在乎自己的需求是否得到了满足；而项目工期是很紧的，自行开发“甘特图绘制包”势必增加其他工作的压力，为什么不采用第三方 SDK 呢？

另一方面，短期内决定采用的第三方“甘特图绘制包”可能并不是最优的，所以并不希望 PM Tool “绑死”在特定“甘特图绘制包”上。

基于以上分析，架构师会决定：采用第三方 SDK，但会自主定义“甘特图绘制接口”将 SDK 隔离。如图 1-8 所示。

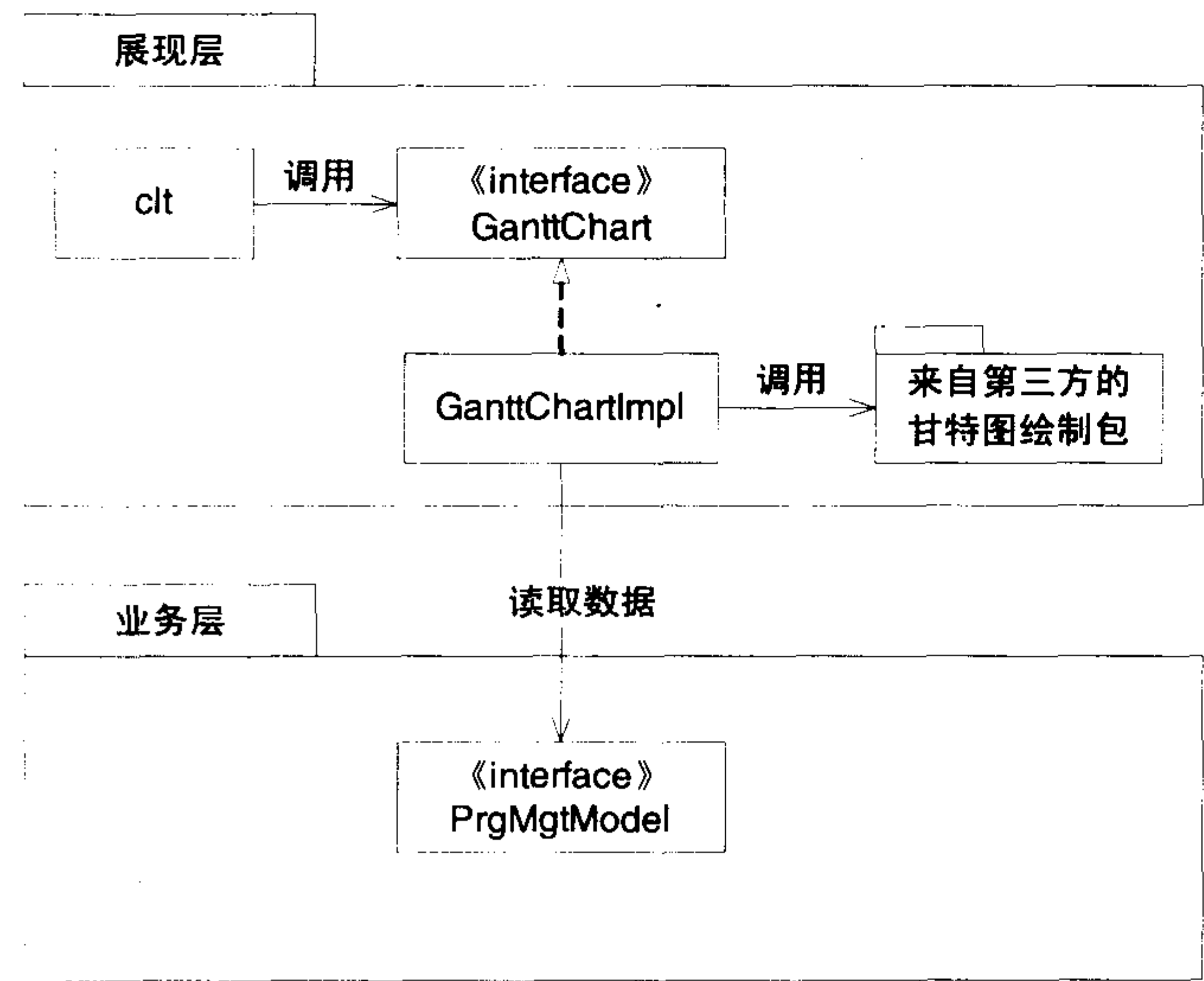
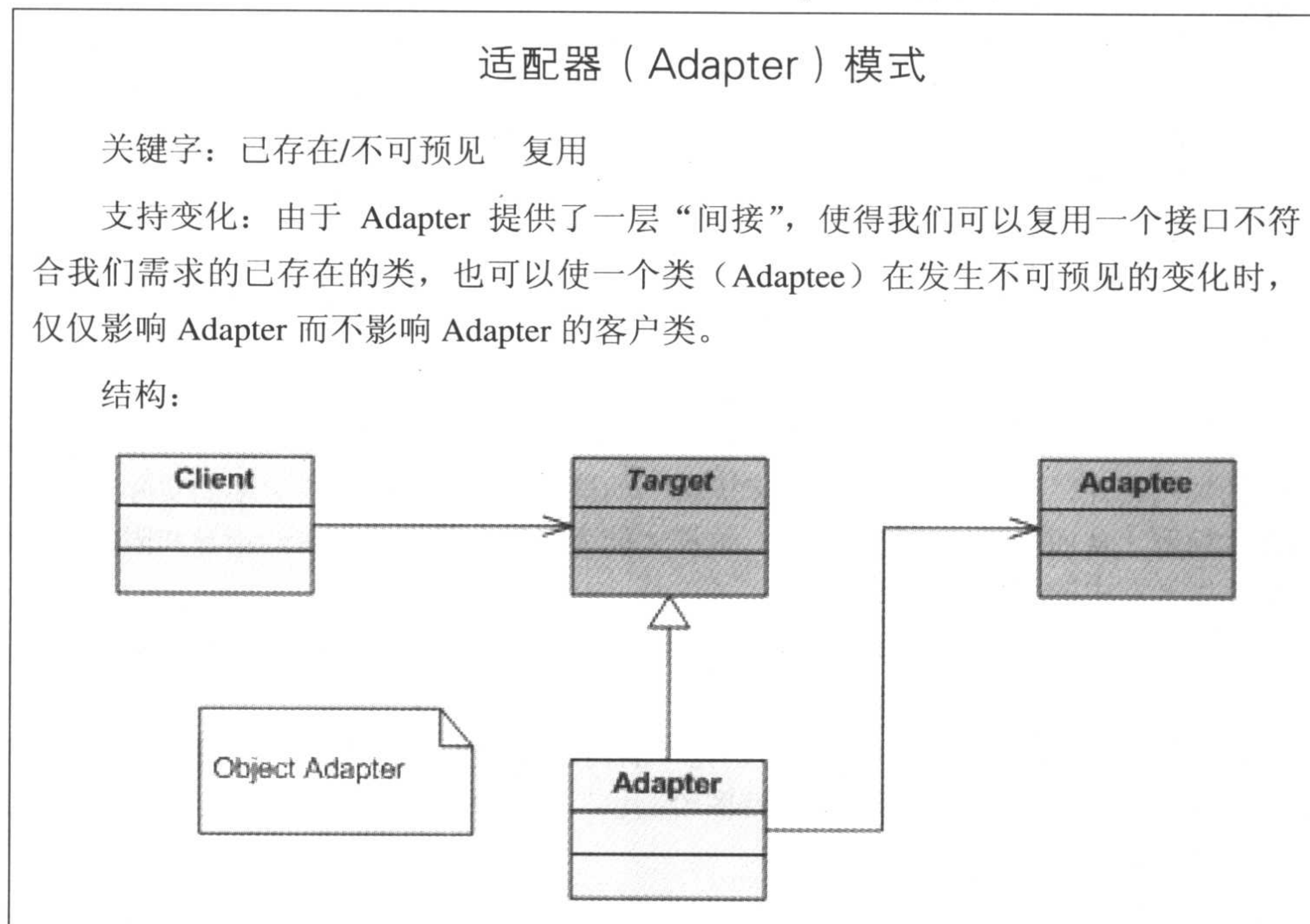


图 1-8 和技术相关的架构方案

有的读者应该已经看出来了，上述设计中采用了 Adapter 设计模式。



### 1.5.2 软件架构概念的体现

有关 PM Tool 的案例故事先讲到这儿，虽然其中仅涉及到架构设计方案的一小部分，但仍然可以体现软件架构的概念——对组成派和决策派的架构概念都有体现。

先说组成派的架构概念，它强调软件架构包含了“计算组件及组件之间的交互”。组件体现在哪里呢？

在图 1-7 所展示的设计中，“业务层”和“展现层”就是两个组件；当然，这两个组件粒度很粗，并且完全是黑盒。

到了图 1-8 所展示的设计中，黑盒虽然没有完全变成白盒，但支持 MVC 协作机制的一部分关键类已经明确——PrgMgtModel、GanttChart 和 GanttChartImpl 都是粒度较细的组件，可以说，“业务层”和“展现层”两个组件在某种程度上已从黑盒变成了灰盒，从而能提供更具体的开发指导。

那么，软件架构概念中所说的“交互”体现在哪里呢？

对于图 1-7 所展示的设计，“业务层”和“展现层”两个粗粒度组件之间的交互为：展现层从业务层“读取数据”。



“读取数据”这一交互到了图 1-8 的设计依然存在，但此职责已经“具体落实”成了“GanttChartImpl 从 PrgMgtModel 读取数据”。另外，图 1-8 的设计中，两个“调用”关系也是软件架构的概念中“交互”的具体例子。

由此看来，组成派软件架构概念完全是对架构设计方案的忠实概括，只不过有一点儿抽象罢了。

再看看决策派的架构概念，它归纳了架构决策的类型，指出架构决策不仅包括关于软件系统的组织、元素、子系统、架构风格等的几类决策，还包括关于众多非功能需求的决策。图 1-7 所展示的设计那么简单，也包含了设计决策吗？是的，业务层和展现层分离，体现了架构概念中的“软件系统的组织”决策，这一设计决策早已得到了业界的普遍认同。

下面再举个例子，来说明软件架构中的设计决策是如何支持“使用、功能性、性能、弹性、重用、可理解性、经济和技术的限制及权衡，以及美学等”非功能需求的。仅以“弹性”为例吧。为了防止 PM Tool “绑死”在特定甘特图绘制包上，架构设计之时作了如下决策（参见图 1-8）：引入自主定义的 GanttChart 接口，让实现该接口的 GanttChartImpl 转而调用第三方 SDK（其实就是 Adapter 设计模式）。这样一来，架构就有了弹性——当发现功能更强大的甘特图程序包时（或决定直接调用 Java 2D 自行开发甘特图绘制部分时），可以方便地仅更改 GanttChartImpl，而其他组件不用更改（如图 1-9 所示）。

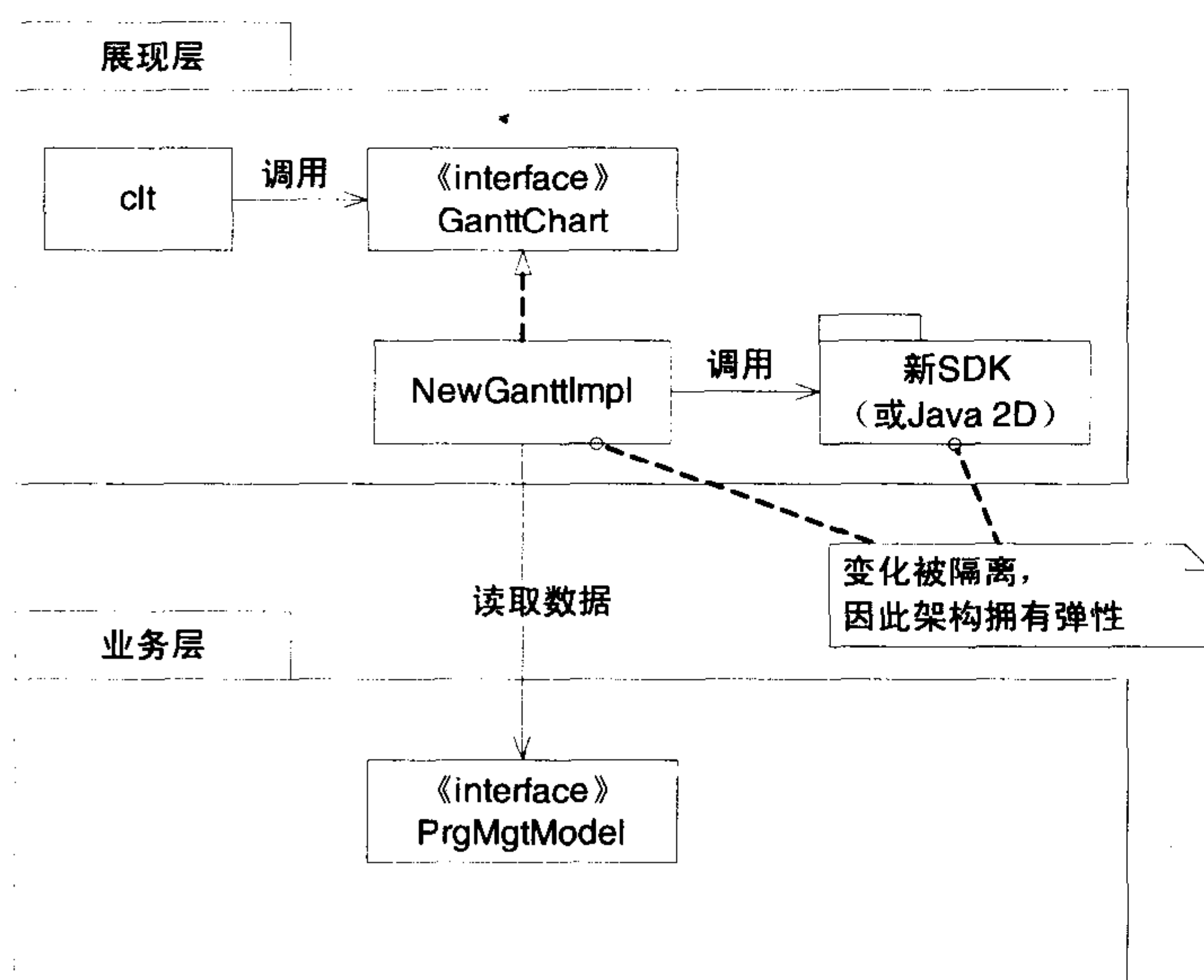


图 1-9 软件架构如何具有弹性

### 1.5.3 重要结论

通过上述分析，我们高兴地看到：组成派和决策派软件架构概念并不矛盾，它们只不过是所站的角度不同罢了；在具体的软件架构实践中，总是同时体现着这两“派”的架构概念。

## 1.6 总结与强调

---

不积跬步，无以至千里。本章仅是一小步，讨论软件架构的基本概念。虽然本书旨在系统地说明软件架构设计的方法与过程，但首先阐明软件架构的概念是大有裨益的，也是非常必要的。

软件架构概念是多样的。虽然软件架构的概念至今依然没有统一，但作为软件架构师，我们不能“揣着手儿”等待，将软件架构的概念总体上分为组成派和决策派，有利于我们理解软件架构概念的精髓。

本章还通过“用案例说话”的方式，说明了两个架构概念流派虽然角度不同，但却相辅相成。我们既应从“架构=组件+交互”的观点中获益，又应运用“架构=重要决策集”的实践经验，这一点对于软件业界的实践者（而不仅仅是理论研究者）尤其重要。



## 第 2 章 子系统、框架与架构

---

复杂性是层次化的。——Frederick P. Brooks, 《人月神话》

自从软件系统首次被分成许多模块, 模块之间有相互作用, 组合起来有整体的属性, 就具有了体系结构。

——张友生, 《软件体系结构的概念》

好的架构必须使每个关注点相互分离.....

——Ivar Jacobson, 《AOSD 中文版》

有两个概念和软件架构的基本概念是相伴相生的。特别是在实践中, 它们总是在软件架构设计过程中出现。

**第一个是子系统的概念。**子系统是随着软件复杂性的增长而日渐重要的一个概念, 它和软件架构密切相关; 软件产业发展到今天, 软件系统的规模也越来越大, 所有软件系统都会被划分为多个模块或子系统进行开发; 当子系统也足够复杂时, 子系统本身的开发也需要经过架构设计这一关。另一方面, 系统整合的趋势日渐强劲, 对于大型企业来讲, 直接规划近一二十年的综合信息系统方案(它是多个相关软件系统组成的“超系统”或称“系统的系统”)也并不鲜见。于是, 软件架构师也应了解软件架构的层次(如软件超系统的架构、软件系统架构、软件子系统架构等)以及不同层次的架构模式(如 SOA 和 MVC 就处在不同的层次上)。

**第二个就是框架。**当前, 基于框架的开发堪称一种文化。为了提高软件开发的“起点”, 以加快开发速度, 提高产品质量, 基于框架进行开发已成为了一种普遍现象。一个项目采用一个或多个第三方框架是很常见的事情, 例如 Spring、Struts、Swing、.NET 和 MFC 等都是流行或曾经流行的框架, 不一而足。了解相关技术领域或业务领域的框架已成为软件架构师的必修课。

本章将结合实践, 讲述子系统和框架在软件架构设计中的地位。

## 2.1 子系统和框架在架构设计中的地位

### 2.1.1 关注点分离之道

好的架构设计必须把变化点错落有致地封装到软件系统的不同部分，为此，必须进行关注点分离。Ivar Jacobson 在《AOSD 中文版》中写道：

好的架构必须使每个关注点相互分离，也就是说系统中的一部分发生了改变，不会影响其他部分。即使需要改变，也能够清晰地识别出哪些部分需要改变。如果需要扩展架构，影响将会最小化。已经可以工作的每个部分都将继续工作。

那么，如何通过关注点分离来达到“系统中的一部分发生了改变，不会影响其他部分”的目标呢？

首先，可以通过职责划分来分离关注点。面向对象设计的关键所在，就是职责的识别和分配。每个功能的完成，都是通过一系列职责组成的“协作链条”完成的；当不同职责被合理分离之后，为了实现新的功能只需构造新的“协作链条”，而需求变更也往往只会影响到少数职责的定义和实现。无论是对象、模块，还是子系统，它们所承担的职责都应该具有高内聚性，否则对象之间的松耦合性就失去了基础，成为空谈。架构模式和设计模式为特定上下文中重复出现的问题提供了通用的职责划分方案。

其次，可以利用软件系统各部分的通用性的不同进行关注点分离。不同的通用程度意味着变化的可能性不同，将通用性不同的部分分离有利于通用部分的重用，也便于对专用部分进行修改。打个比方，一座高楼大厦要想稳固，必须考虑不同部分的热胀冷缩系数的差异，将不同“膨胀比”的部分硬连在一起容易引起“开裂”，这个比喻很好地说明了“稳固的架构”的含义。广为人知的框架技术可以用于分离通用部分，而元模型驱动方法是另一种分离通用性部分的技术。

另外，还可以先考虑大粒度的子系统，而暂时忽略子系统是如何通过更小粒度的模块和类组成的。在实际中，软件架构师常常将系统划分为一组子系统，并为子系统定义明确的接口，其中的细节将随其后的开发工作慢慢展开。这样做可以避免陷入过多的细节当中，所谓“忘却是一种能力”，就是指架构师必须有在更高层面思考的能力（第 19 章将以继承为例说明如何突破 OOP 思维的限制）。

图 2-1 总结了上述的架构设计关注点分离原理。可以说，根据职责分离关注点、根据通用性分离关注点、根据不同粒度级别分离关注点是 3 种位于不同“维度”的思维方式，所以在实际工作中必须综合运用这些手段。



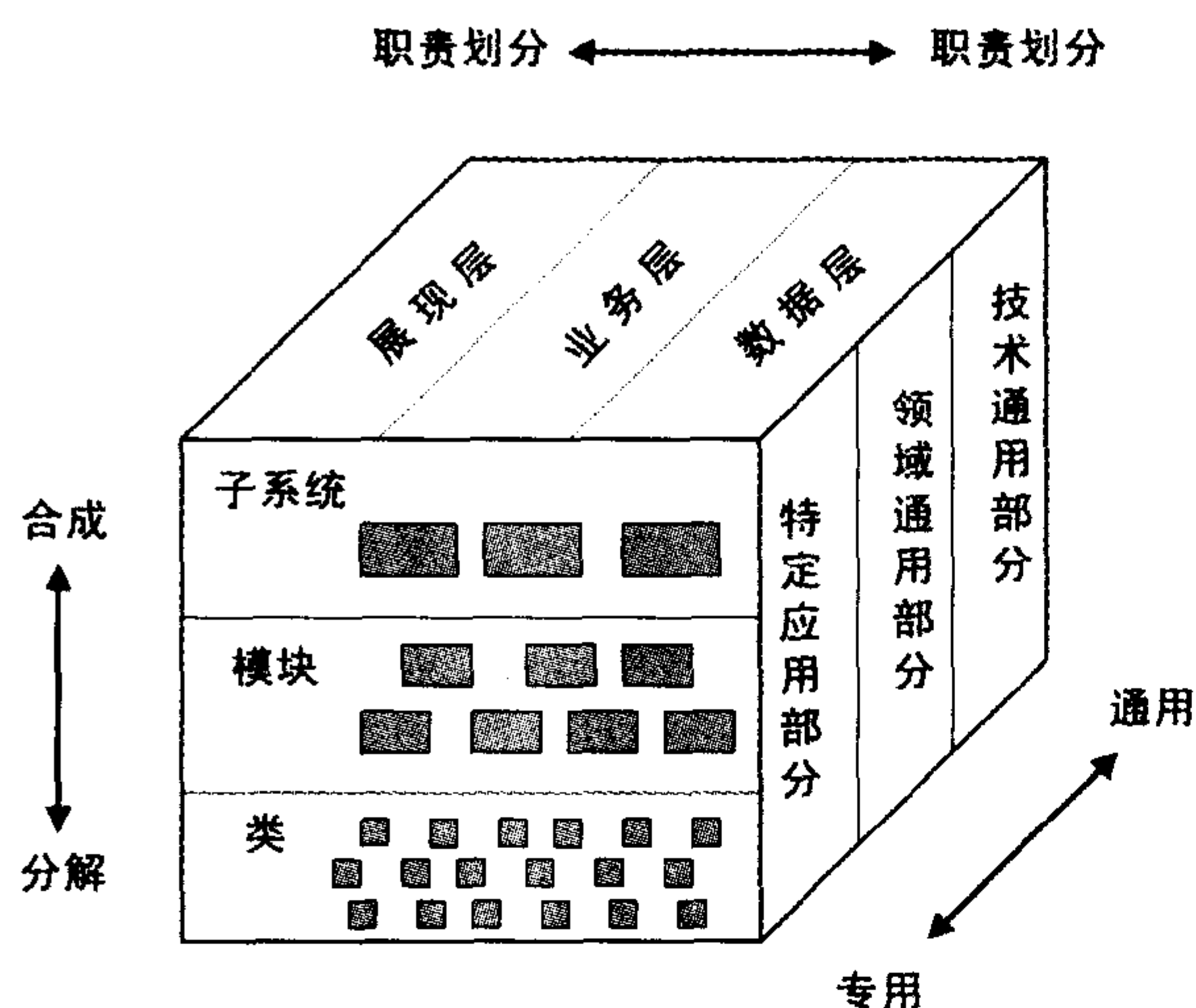


图 2-1 架构设计关注点分离原理

### 2.1.2 子系统和框架在架构设计中的地位

软件界是新名词制造工厂，这可能是其他任何产业都难以望其项背的。但在这背后，深藏着的是相对稳定的“解决之道”。根据我们上一节讲述的关注点分离原理，归纳了一些流行技术所处的位置，如图 2-2 所示。

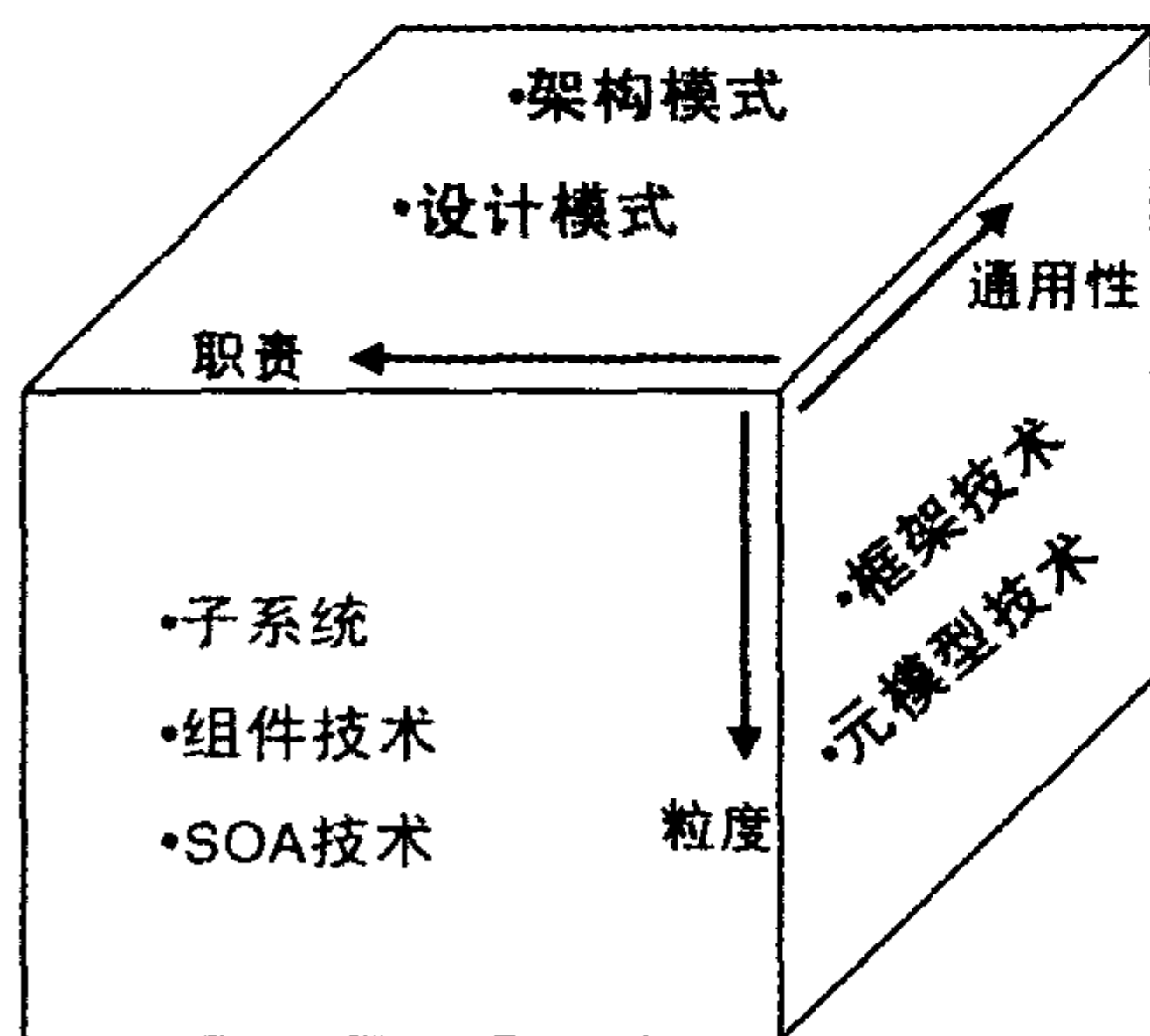


图 2-2 技术谱

例如，无论是架构模式还是设计模式，重点关注的都是如何提供一个“协作模型”，这个协作模型通过明确协作中不同角色所担负的职责，达到“为特定上下文中的问题提供解决方案”的目的。来看看抽象工厂（Abstract Factory）设计模式，它是常见的设计模式之一。图 2-3 以“上下文-问题-解决方案”的形式总结了抽象工厂设计模式：我们遇到的设计问题是如何实现一系列对象的实例化，并且问题所处的上下文是“不同的应用场景需要不同系列的对象实例”；最终，

抽象工厂的解决方案是“为创建系列对象提供统一接口，为如何实际创建提供不同实现”。显然，如果没有“不同的应用场景需要不同系列的对象实例”上下文限制，我们的设计可能仅仅是个普通的对象工厂。

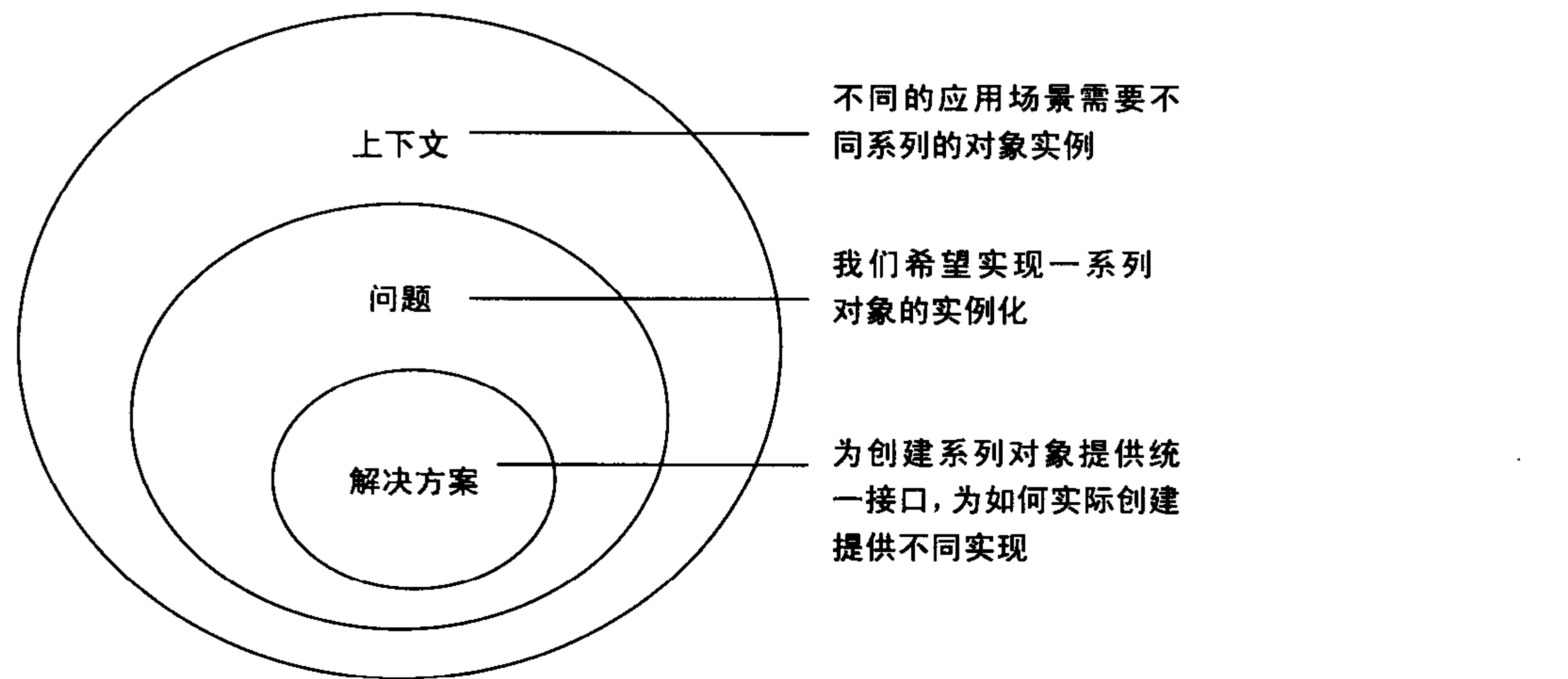
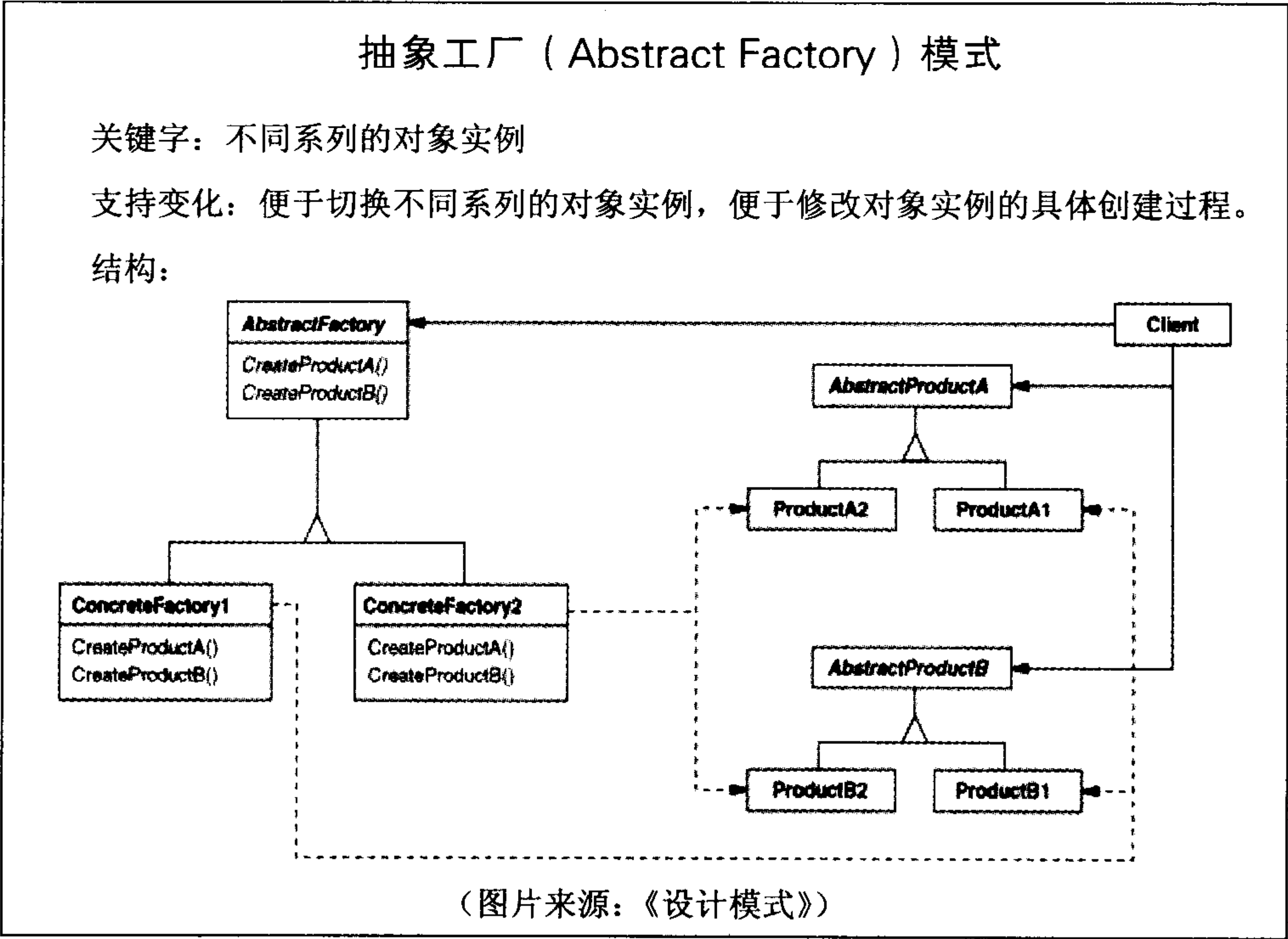


图 2-3 抽象工厂设计模式



于是，我们就不难理解本章将重点讨论的子系统和框架技术在架构设计中的重要地位了。例



如，现在的软件开发越来越倚重框架的使用，因此选择何种框架，每个框架在整个架构中处在什么位置，都成为软件架构设计的重要环节。Ivar Jacobson 就曾指出，“设计应该把类库和框架的用法反映出来”。框架技术有助于把通用关注点和专用关注点分离开来，结果是带来了更好的易修改性和可重用性。如图 2-4 所示，框架支持我们引入一个全新的关注点分离维度，并且它和分层架构有很好的结合。

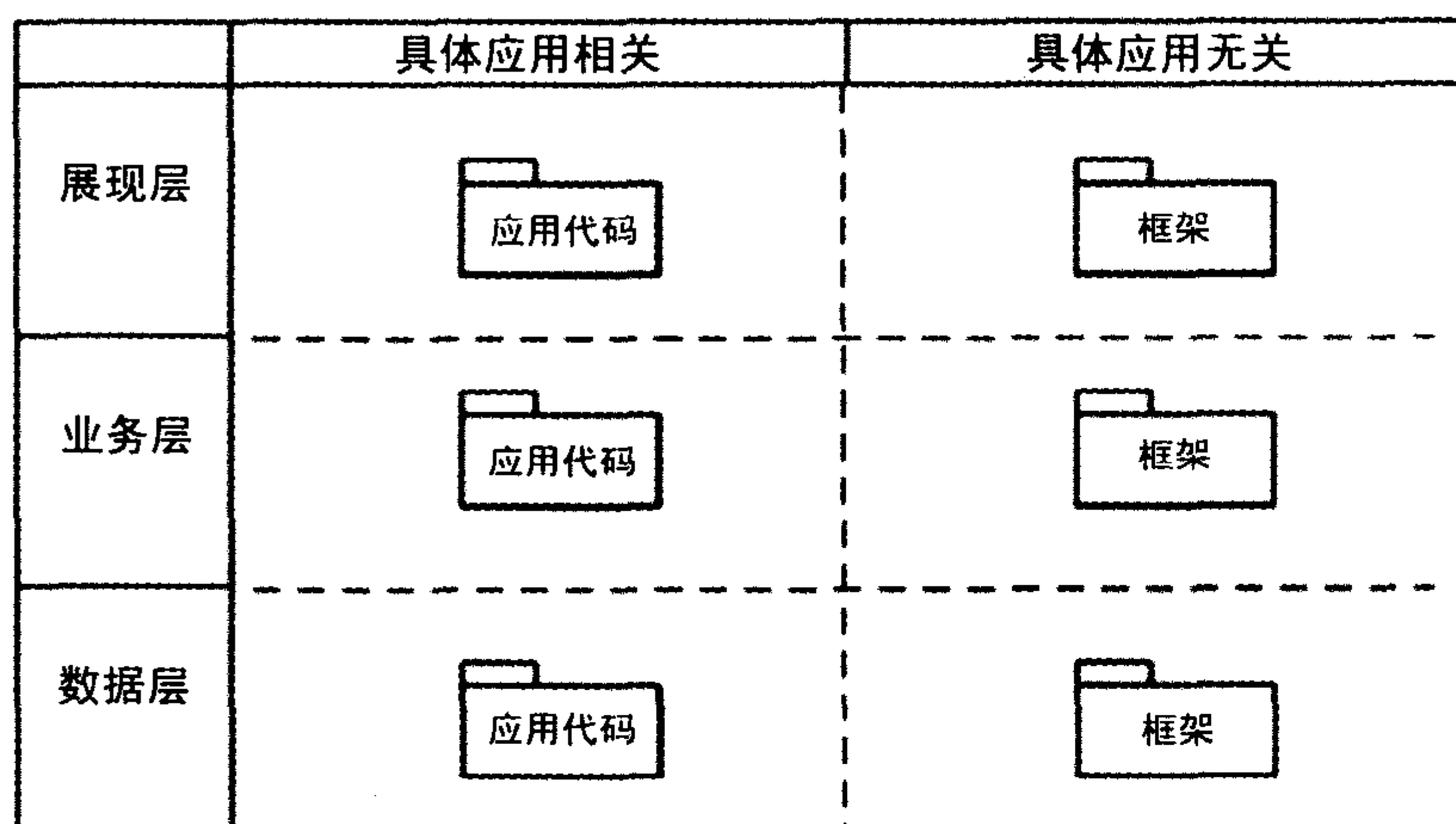


图 2-4 框架引入一个全新的关注点分离维度

## 2.2 子系统与软件架构

了解子系统与软件架构的关系有两个方面的意义。

从解决问题的角度而言，了解子系统与软件架构的关系，有助于避免软件架构设计不足和高来高去的问题（可参考第 8 章）。软件架构要设计到什么程度？为了使软件架构能够为软件开发提供足够的指导和限制，软件架构师应当为复杂的子系统设计架构，而不是保持其黑盒子的状态止步不前。

从经验的运用角度而言，了解子系统与软件架构的关系，有助于充分利用架构设计技能。比如，有经验的软件架构师都知道，分层、MVC 和管道过滤器等架构模式很少单独使用，而是结合使用或根据子系统的层次划分嵌套使用。如图 2-5 所示，就是一个分层架构模式和 MVC 架构模式结合使用的例子。

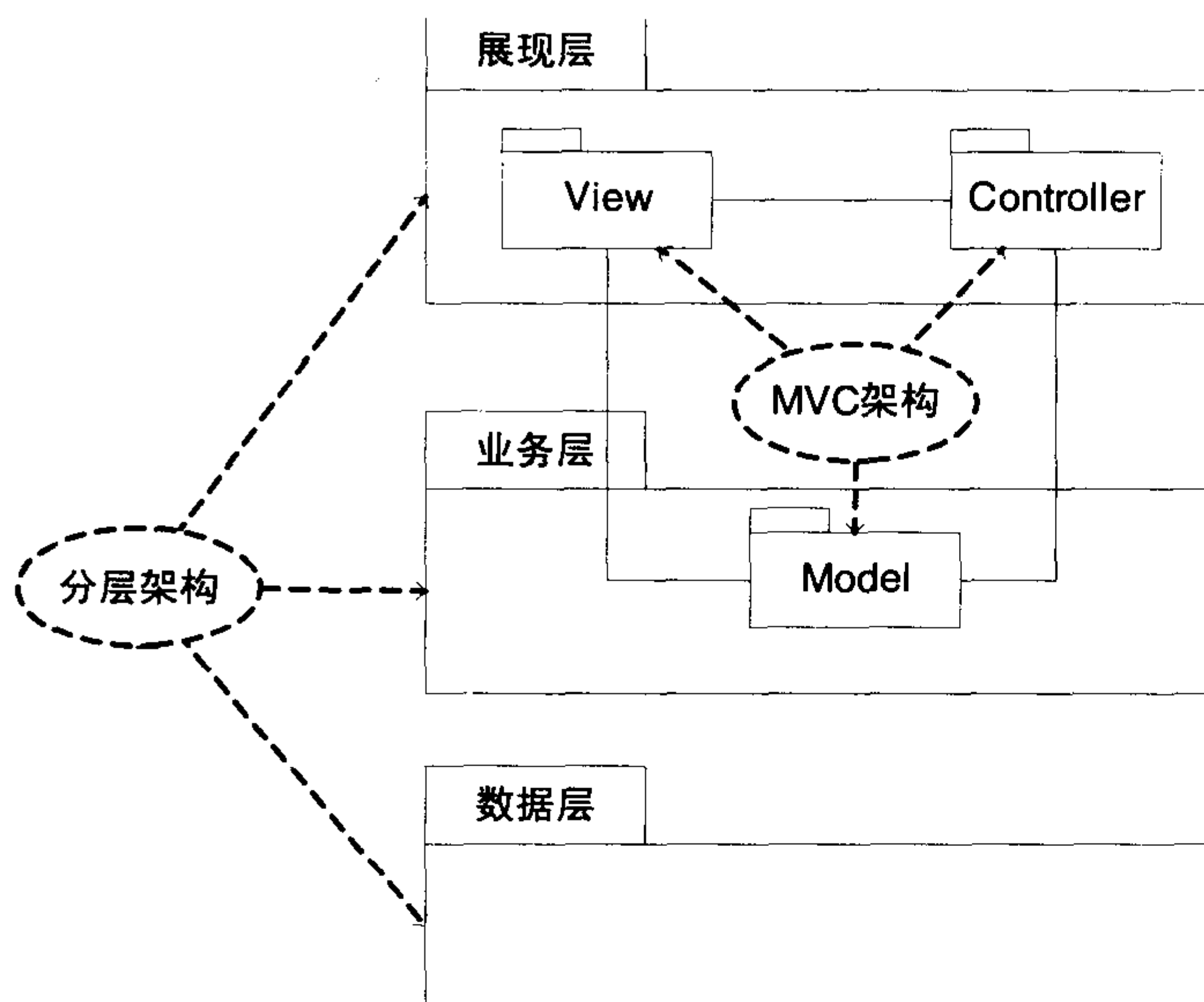


图 2-5 分层架构和 MVC 架构结合使用的例子

### 2.2.1 不同粒度的软件单元

作为软件架构师，必须思考“软件单元是如何组成粒度更大的整体的”这一问题。

在具体的架构实践中，一个软件系统往往首先分解为几个子系统，子系统又继续分解……如图 2-6 所示，它刻画了这样的软件分解场景：一个系统，由 3 个子系统组成；每个子系统，又由组成它的多个类来实现。子系统可以分配到开发组，每个类可以分配给具体的工程师实现。

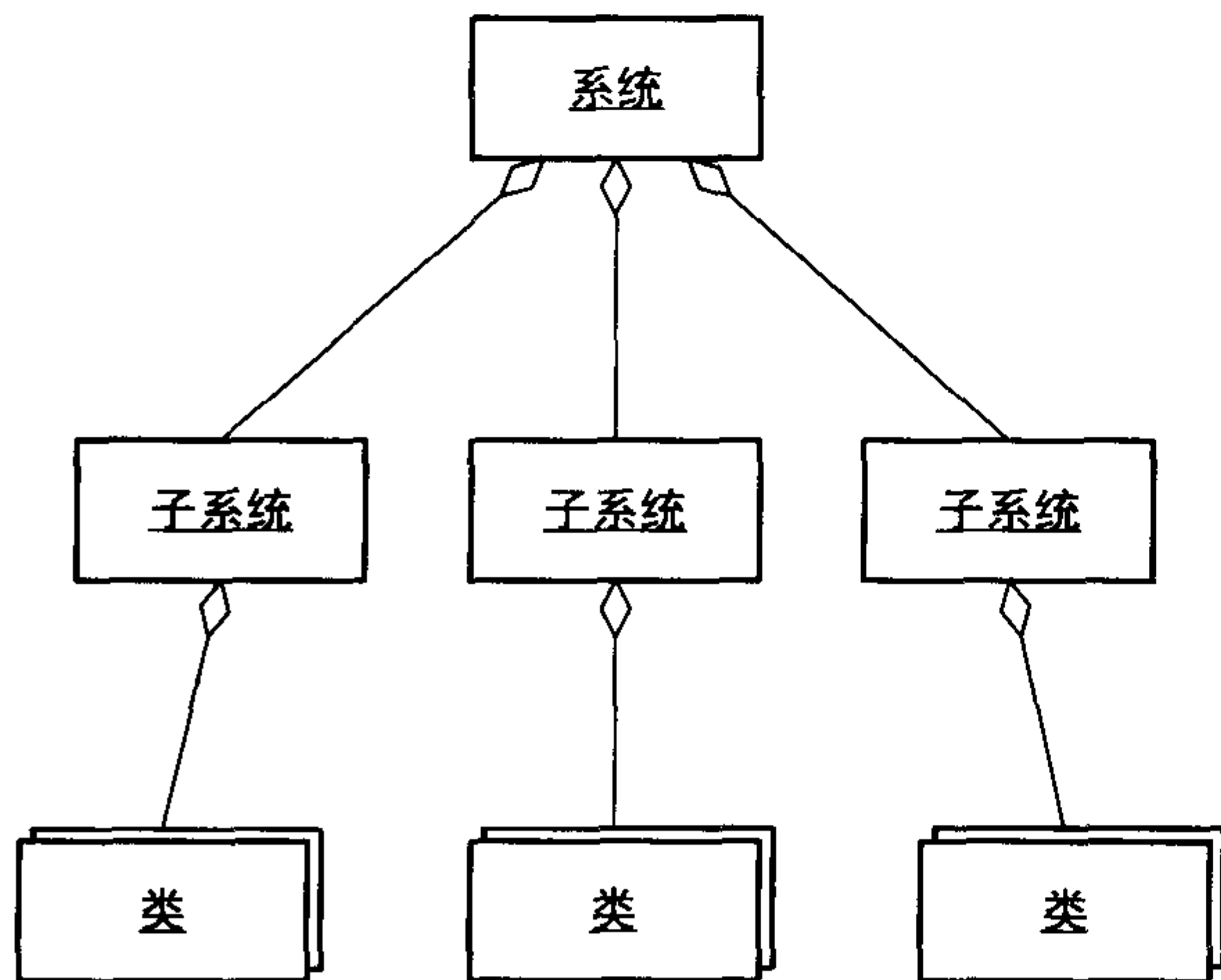


图 2-6 一个软件分解场景

更广泛而言，构成软件的单元具有不同的粒度等级。对于面向对象的软件开发而言，经常有



这些软件单元：

- 粒度最小的单元通常是“类”
- 几个类紧密协作形成“模块”
- 完成相对独立的功能的多个模块构成了“子系统”
- 多个子系统相互配合才能满足一个完整应用的需求，从而构成了软件“系统”
- 一个大型企业往往使用多套系统，多套系统通过互操作形成“集成系统”

类、模块、子系统、系统、集成系统，都是软件单元的具体形态，只不过粒度不同罢了。软件系统越复杂，不同粒度的分解层次就越多；比如，有的组织会引入“分系统”的概念，形成“系统—分系统—子系统—模块”的分级体系。

2.2.2 子系统也有架构

所谓系统，是指由多个元素组成的逻辑实体，它完成一组特定的目标或担负一定的职责。系统可以仅包含软件，也可以仅包含硬件，或者是两者都包含。

子系统是特殊的系统——只不过在特定的上下文中，这个系统作为更大的系统的一部分出现。

系统需要架构设计，而子系统如果足够复杂，则也需要架构设计。图 2-7 揭示了这一点：子系统作为特殊的系统，它也是由多个元素组成的整体，而架构规定了它是如何划分为多个部分的，以及规定了多个部分之间的交互机制和交互接口。

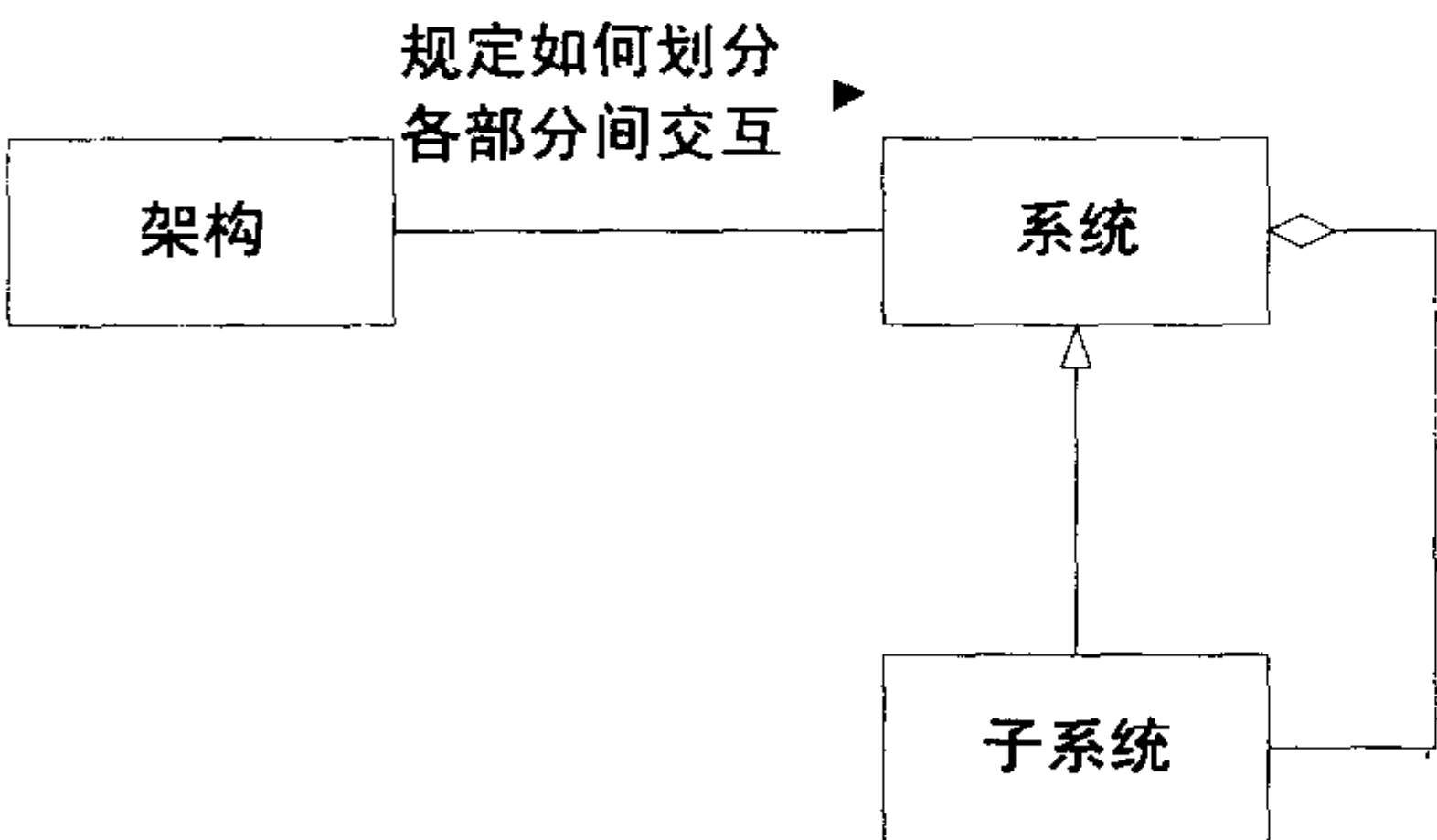


图 2-7 子系统也有架构

2.2.3 子系统不同，架构不同

另外，不同类型的软件系统需要不同的软件架构设计，这似乎是很多人都理解的道理；但有时候，一个系统的不同子系统也应当有不同的软件架构。

举个例子。相信不少读者了解 Martin Fowler 所著的《企业应用架构模式》中介绍的事务脚本模式（Transaction Script）、领域模型模式（Domain Model）等“领域逻辑模式”。在实际的架

构设计当中，这些模式的运用并不是“放之各子系统而皆准”的。例如，一个采用了分层架构的软件系统，它可能包含了报表、拓扑显示等子系统，这些子系统会有自己的内部架构吗？

图 2-8 所示的示意图给出了结论：拓扑子系统适宜采用领域模型架构模式，而报表子系统则应采用事务脚本架构模式。

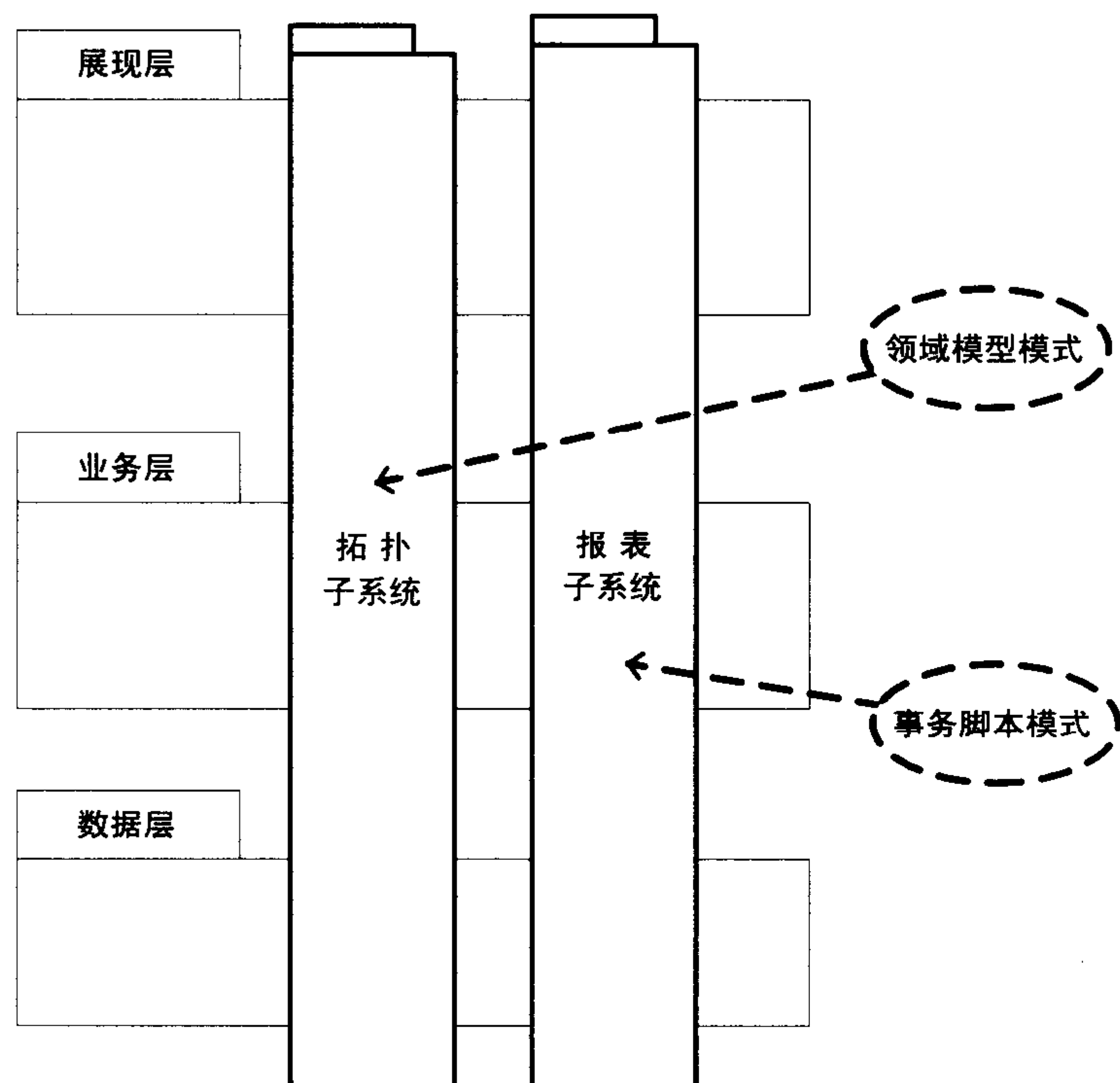


图 2-8 不同子系统采用不同软件架构的例子

对于此例，如果你熟悉网管软件，你可以想象这是个网络设备拓扑图的显示子系统；如果你熟悉 UML 建模，你可以想象这是个 UML 建模工具……拓扑子系统的业务逻辑复杂，诸如图元的排列、连接、移动、覆盖、复制和删除等问题涉及到不同的规则，应当充分利用对象模型的优势来解决这些问题。

至于此例中的报表子系统，其业务逻辑相对比较简单，通过 SQL 语句从数据库中提取数据非常方便，并且可以利用 SQL 语句进行一些统计和查找计算，因此宜于采用事务脚本模式（转而采用领域模型模式无疑是自找麻烦）；当然，报表子系统采用事务脚本模式对提高性能也大有好处。



### 2.2.4 不同实践者眼中的粒度

另外，在真实的软件实践中，还有一个问题不可忽略：粒度的相对性。也就是说，同一个软件单元，在不同场景下我们会以不同的粒度看待它。

我们举个具体的例子。这个例子看似极端，因为在大多数开发者看来，一个 ActiveX 控件是再小不过的一个单元了，但它对控件的开发方来说可能是个子系统。诚然，一个 VB 编程者可以把 ActiveX 控件拖过来就用，完全把它当成原子级的软件单元；而对于开发方，这个 ActiveX 控件（比如报表控件、音频编辑控件）就是一个蛮复杂的“子系统”，从需求分析到架构设计再到编码测试，所花费的时间一点儿也不比其他类型的软件项目少。

这就好比，对于除菌皂“研制人员”而言，肥皂和细菌都是复杂体：“研制人员”需要分析细菌的结构，设计肥皂的成分。另一方面，对于用除菌皂“洗手的人”而言，它们并不关心肥皂和细菌的内部。

由此看来，是复杂事物还是简单事物，要视对谁而言。在某个实践者眼中的最基本的组件，对另一个实践者而言有可能位于相当高的抽象层次上。对此，《Systems Architecting》一书的作者 Rehtin 就曾指出，“所有系统都有子系统，而所有系统都是更大系统的子系统”。此话再明确不过地揭示了一点：实践的不同场景使得软件组成单元具有递归性。

因此，我们应当立足实践对待软件单元的粒度问题（从第 1 章关于“软件架构是一系列有层次性的决策”的讲解中也可以理解本问题）：

- 当进行高一级的架构设计时，子系统就是一个原子单元、一个黑盒；
- 当进行子系统本身的设计时，它才变成一个结构复杂的白盒；
- 第三方组件一般被看作原子单元，只需关心其对外接口即可——此时我们是使用它的服务；
- 但当使用来自第三方的框架作为某一级系统或子系统的架构基础时，则应当仔细研究其结构——这是因为此时是使用它的结构而不仅仅是服务。

总之，懂得了“细节是相对而言的”这一点，软件架构师才能够游刃有余地根据情况忽略应该被忽略的细节，抓住设计大局。

## 2.3 框架与软件架构

### 2.3.1 框架的概念

我总结的框架的定义是：框架是可以通过某种回调机制进行扩展的软件系统或子系统的半成品。

首先，框架是半成品，这是它和其他所有软件组件的本质区别。这涉及到“软件重用”的一对内在矛盾：“重用几率”大小和“重用所带来的价值量”大小之间的矛盾。简言之，软件单元的粒度越大，则重用所带来的价值量越大，但重用几率越小；反之，粒度小的软件单元被重用的几率越大，则重用所带来的价值量就越小。框架的智慧就在于此：为了追求重用所带来的价值量最大化，将容易变化的部分封装成扩展点，并辅以回调机制将它们纳入框架的控制范围之内，从而在兼顾定制开销的同时使被重用的设计成果最多。

下面，来解释一下“某种回调机制”的含义。框架并不一定必须用面向对象编程语言实现，正如第2.5.4节“如何实现框架中的扩展点”中将详细讲解的，如C语言等传统编程语言可以通过函数指针作为参数来实现回调机制，而面向对象编程语言中利用抽象方法（C++中称为虚函数）支持回调机制——无论是抽象类还是接口都利用了抽象方法。

Frank Buschmann 等人在《面向模式的软件体系结构（第一卷）》中为框架所下的定义，非常重视框架与架构的关系：

框架是一个可实例化的、部分完成的软件系统或子系统，它为一组系统或子系统定义了架构，并提供了构造系统的基本构造块，还为实现特定功能定义了可调整点。在面向对象环境中，框架由抽象类和具体类组成。（A framework is a partially complete software (sub-) system that is intended to be instantiated. It defines the architecture for a family of (sub-) systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made. In an object-oriented environment a framework consists of abstract and concrete classes.）

相比之下，Erich Gamma 等人在《设计模式》中为框架所下的定义则循循善诱，符合面向对象开发者的口味：

（框架是）一组相互协作的类，形成某类软件的一个可复用设计。框架将设计划分为一组抽象类，并定义它们各自的责任和相互之间的协作，以此来指导体系结构级的设计。开发者通过继承框架类中的类和组合其实例来定制该框架以生成特定的应用。

### 2.3.2 架构和框架的区别

笔者发现，人们对软件架构存在非常多的误解，其中一个最为普遍的误解就是：将架构（Architecture）和框架（Framework）混为一谈。

一图胜千言，图2-9切中肯綮地点出了架构和框架的区别。一句话，框架是软件，架构不是软件。

框架是一种特殊的软件，它并不能提供完整无缺的解决方案，而是为你构建解决方案提供良好的基础。框架是半成品。典型地，框架是系统或子系统的半成品；框架中的服务可以被最终应用系统直接调用，而框架中的扩展点是供应用开发人员定制的“可变化点”。

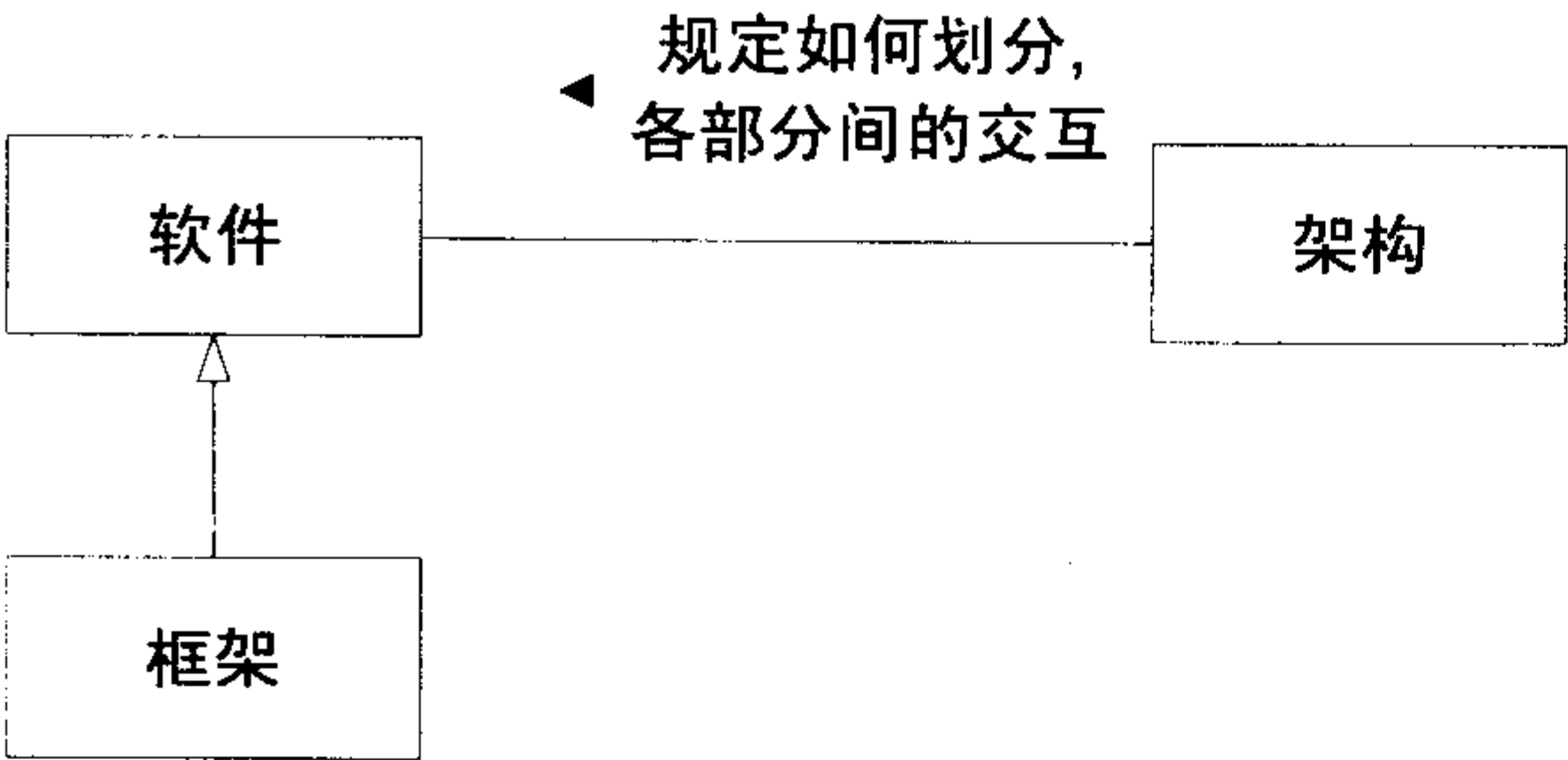


图 2-9 架构和框架的区别

软件架构不是软件，而是关于软件如何设计的重要决策。软件架构决策涉及到如何将软件系统分解成不同的部分、各部分之间的静态结构关系和动态交互关系等。经过完整的开发过程之后，这些架构决策将体现在最终开发出的软件系统中；当然，引入软件框架之后，整个开发过程变成了“分两步走”，而架构决策往往会体现在框架之中。或许，人们常把架构和框架混为一谈的原因就在于此吧！

我们不能指着某些代码，说这就是软件架构，因为软件架构是比具体代码高一个抽象层次的概念。架构势必被代码所体现和遵循，但任何一段具体的代码都代表不了架构。

2.3.3 架构和框架的联系

框架技术和架构技术的出现，都是为了解决软件系统日益复杂所带来的困难而采取“分而治之”思维的结果——先大局后局部，就出现了架构；先通用后专用，就出现了框架。图 2-10 很好地揭示了这一点。架构是问题的抽象解决方案，它关注大局而忽略细节；而框架是通用半成品，还必须根据具体需求进一步定制开发才能变成应用系统。

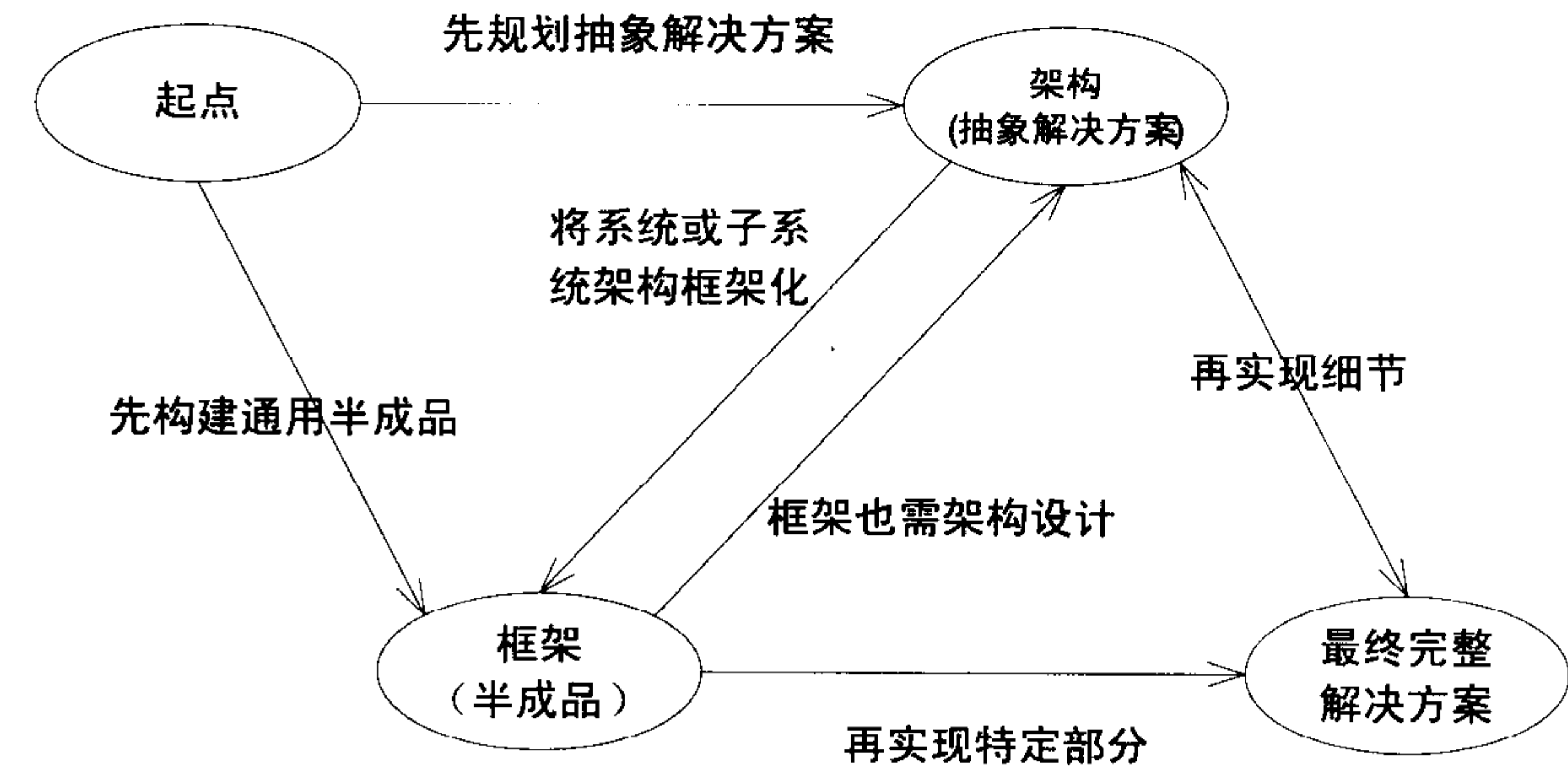


图 2-10 框架和架构的关系



简而言之，框架和架构的关系可以总结为两句话：（1）为了尽早验证架构设计，或者出于支持产品线开发的目的，可以将关键的通用机制甚至整个架构以框架的方式进行实现；（2）业界（及公司内部）可能存在大量可供重用的框架，这些框架或者已经实现了软件架构所需的重要架构机制，或者为未来系统的某个子系统提供了可扩展的半成品，所以最终的软件架构可以借助这些框架来构造。

### 2.3.4 框架也有架构

框架作为软件系统或子系统的半成品，其设计开发过程从总体上来说和系统开发非常类似（是刻意做成“半成品”的，而不是“还没有做完”），框架也是通过架构设计、详细设计、实现和测试开发出来的。当采用面向对象技术时，框架是一组类和接口，当然还可以包含资源文件和配置文件等等。框架可以很复杂，可以包含成百上千个类，可以划分模块和子系统。至于框架的具体开发过程，可以参考 2.5.3 小节“框架的开发过程”。

总之，框架也有架构。图 2-9 所示的类图，其语义也表明了这一点。

## 2.4 超越概念：立足实践理解架构

通过本章前面的讨论，我们了解到：

- 软件系统是由不同粒度的软件单元层层递归构成的，如子系统、模块、类；
- 由于在实践中所处的位置不同，同一个软件单元在不同实践者眼中的粒度可能不同，如实现 Zip 算法的实用类，它的开发者至少认为它是个复杂模块；
- 子系统也有架构，如报表子系统；
- 即使是同一系统内部，子系统不同，所采用的架构也有可能不同，如报表子系统采用事务脚本架构，而拓扑子系统采用领域模型架构模式；
- 框架和架构既有区别又有联系，前者是复合组件特例，后者是复合组件的大局设计；
- 框架也需要架构设计，如 Struts 作为著名框架自然有其架构；
- 反过来，可以通过架构框架化达到“架构重用”的目的，如很多人都在用 Spring 框架提供的控制反转和依赖注入来构建自己的架构。

本节主要是总结软件架构概念的内涵，并理解软件架构的适用范围（即外延）。

### 2.4.1 理解架构

真实的软件其实是“由组件递归组合而成”的：

- 组件的粒度可以很小，也可以很大；任何粒度的组件都可以组合成粒度更大的整体。即所谓的粒度多样性问题；

- 组件粒度的界定，必须在具体的实践上下文中才有意义；你的大粒度组件，对我而言可能是原子组件。即所谓的粒度相对性问题。

由此，我们不能不联想到 Composite 模式。Composite 模式用于表示“部分—整体”的层次结构，可以用来描述软件的递归组合的本质。图 2-11 运用了 Composite 模式来刻画更加真实的软件。

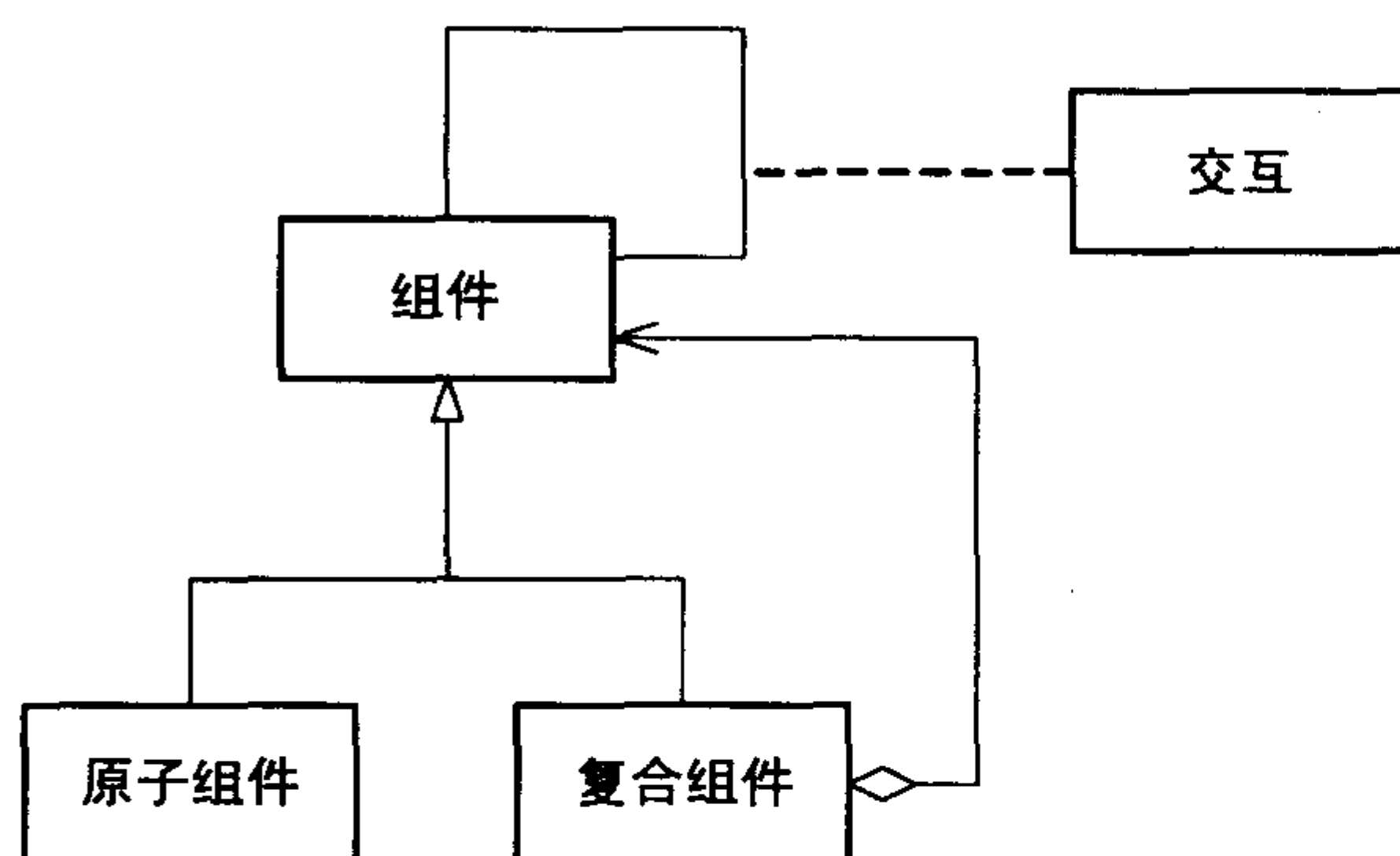


图 2-11 借助 Composite 模式刻画真实的软件

本图借助 UML 类图的语法，表达了一般意义上的“组件合成”场景：组件分为原子组件和复合组件两种；在特定的实践上下文中，原子组件是不可再分的；复合组件是由其他组件（既可以是原子组件，又可以是复合组件）组合而成的；无论是原子组件还是复合组件，它们之间都可以通过交互来完成更复杂的功能。

是时候为“软件架构”找准位置了。

答案看上去惊人的简单，如图 2-12 所示。此图可以用一句话概括：“作为复合整体的软件单元才有架构，架构规定了它如何被设计的重要决策”。

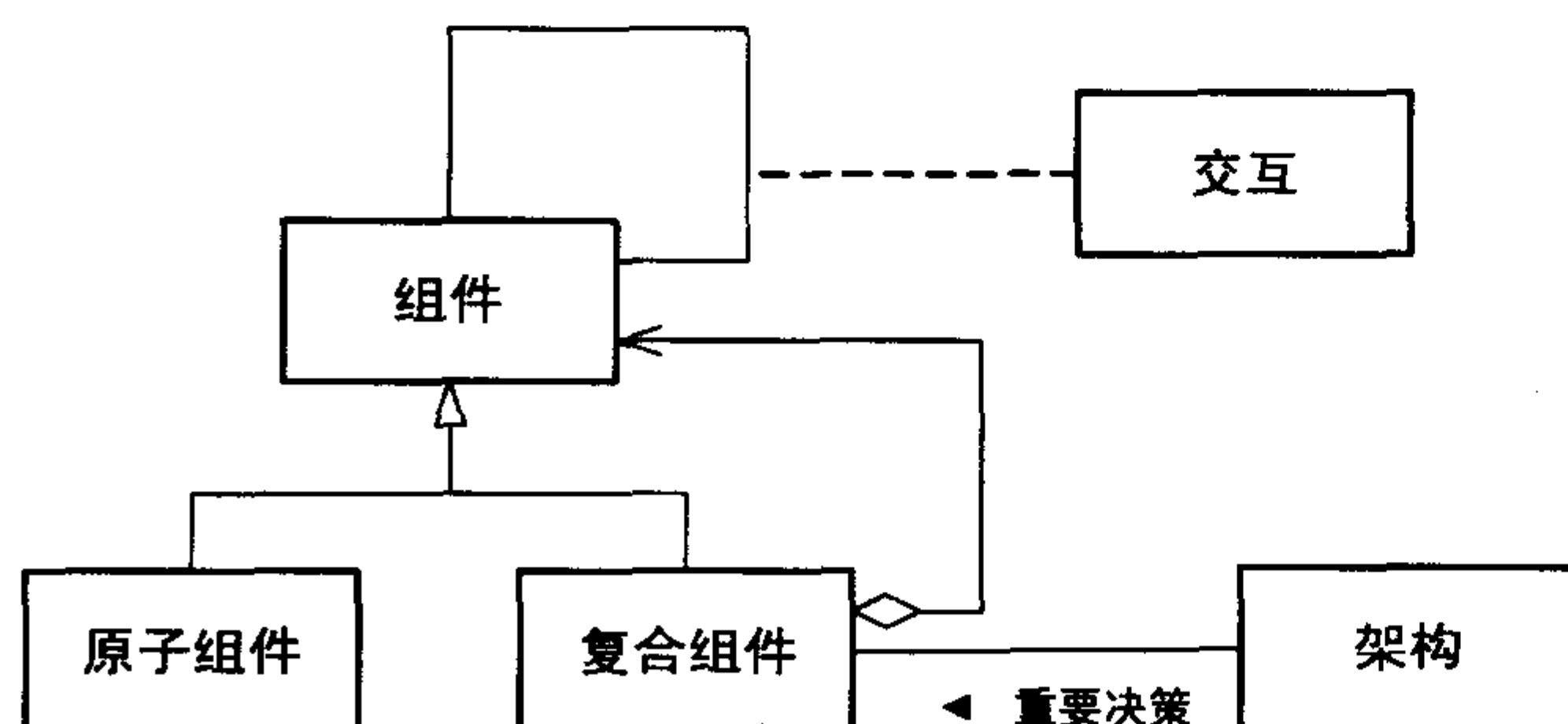


图 2-12 任何复杂整体都有架构

软件架构并不是所有的组件内部的设计都不关心，而是仅仅不关心“在架构设计阶段没有必要进一步分解”的软件单元的内部细节；另外，也不是要将所有设计决策事无巨细地落实，而是

重点关注“重要决策”。软件架构设计的决策范围，应该着重放在“影响全局”的设计上，而不是关注所有设计细节。对此，Len Bass 等人在《软件架构实践》一书中已经明确指出：

架构中包含了关于各元素应如何彼此相关的信息。也就是说，架构必须省略各元素中与交互无关的某些信息。因此，架构首先是对系统的抽象，该抽象去除了不影响它们如何使用、其他元素如何使用以及如何与其他元素关联或交互的细节。在几乎所有的现代系统中，各元素都是通过接口实现交互的，而这些接口又将各元素的细节划分为公有和私有两大类。根据这种划分，架构属于公有部分，而私有部分——即仅与内部具体实现有关的细节——是不属于架构的。

由此可见，软件系统架构关注的是涉及元素之间如何交互的大局，而必须将局部性的细节忽略。其实，关注大局，把握整体，不仅仅是软件系统架构学科的主题，还是所有系统科学研究的对象，钱学森就说过：“什么叫系统？系统就是由许多部分组成的整体，所以系统的概念就是要强调整体，强调整体是由相互关联、相互制约的各个部分所组成的。”

推广开去，其实任何作为复合整体的复杂事物都可能有架构，比如一本书、一幢建筑物。那本“永不褪色的经典”《如何阅读一本书》中就说：“每一本书的封面之下都有一套自己的骨架（Every book has a skeleton hidden between its boards）。”

2.4.2 回到实践

虽然我们最常听到的说法是“软件系统的架构”，但其实未必是完整的软件系统才有架构。在实际的软件实践中，系统、子系统和框架往往都需要进行架构设计。如图 2-13 所示。

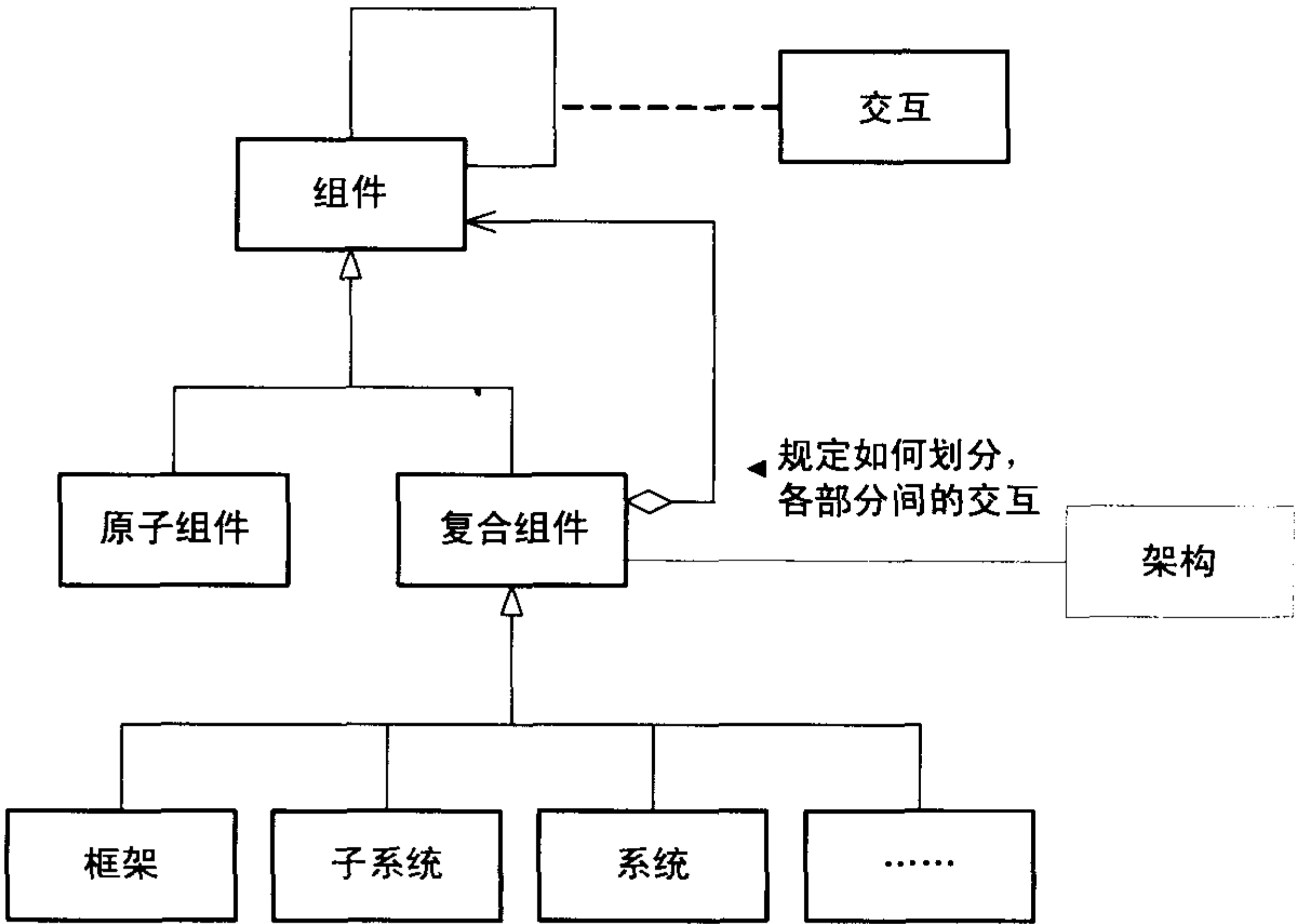


图 2-13 为“软件架构”找准位置



例如，航空航天领域的系统往往极为复杂，这样一来，总的系统需要配备系统架构师，子系统有时也会分别单独配备架构师。

又例如，著名的用友华表 Cell 组件是标准的报表处理 ActiveX 组件，它提供几百个编程接口。虽然在用户看来它只是“开盒即用”的 ActiveX 组件，但这样一个提供强大的制表能力、拥有丰富的单元格类型的报表解决方案，当然需要精心的架构设计。

再例如，随着面向服务架构（SOA，Service Oriented Architecture）被越来越多的人所接受，基于组件的软件工程（CBSE，Component Based Software Engineering）也为更多的人所认识。在此种情况下，整个系统的架构模式是 SOA，而每个组件本身也有自己的架构设计——在实践中不了解这一点会很危险。

## 2.5 专题：框架技术

接下来，对框架技术进行专门讨论。

### 2.5.1 框架 vs. 类库

对面向对象开发而言，类库和框架有很多共同之处，但它们确实又是不同的。通过比较框架和类库的区别，可以更深入地理解框架的概念和内涵。如图 2-14 所示，框架是一种介于类库和应用系统之间的概念。

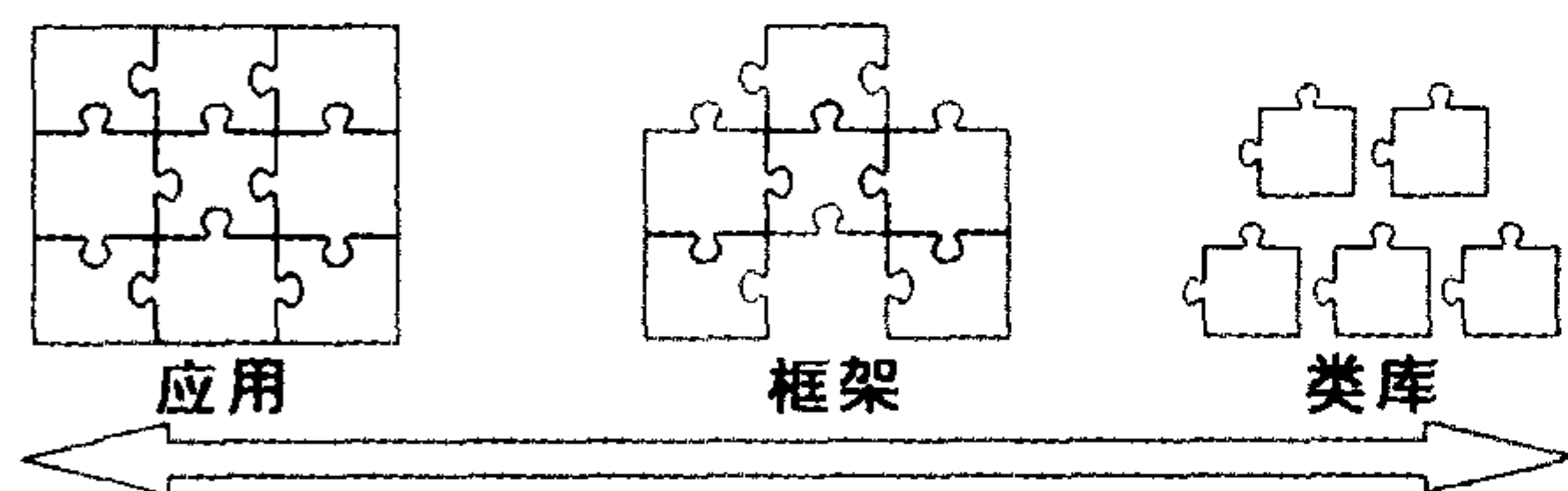


图 2-14 框架介于类库和应用系统之间

类库是类的集合，这些类之间可能是相互独立的。应用开发者希望使用任何一个类时可以直接调用它，而不必再写一个。与类库相比，框架和类库有着相似的形式，即框架也往往是类的集合；但不同之处在于，框架中的各个类并不是孤立的，而框架中的业务逻辑代码是将不同的类“连”在一起，在它们之间建立协作关系。图 2-15 很好地表达了上述意思。

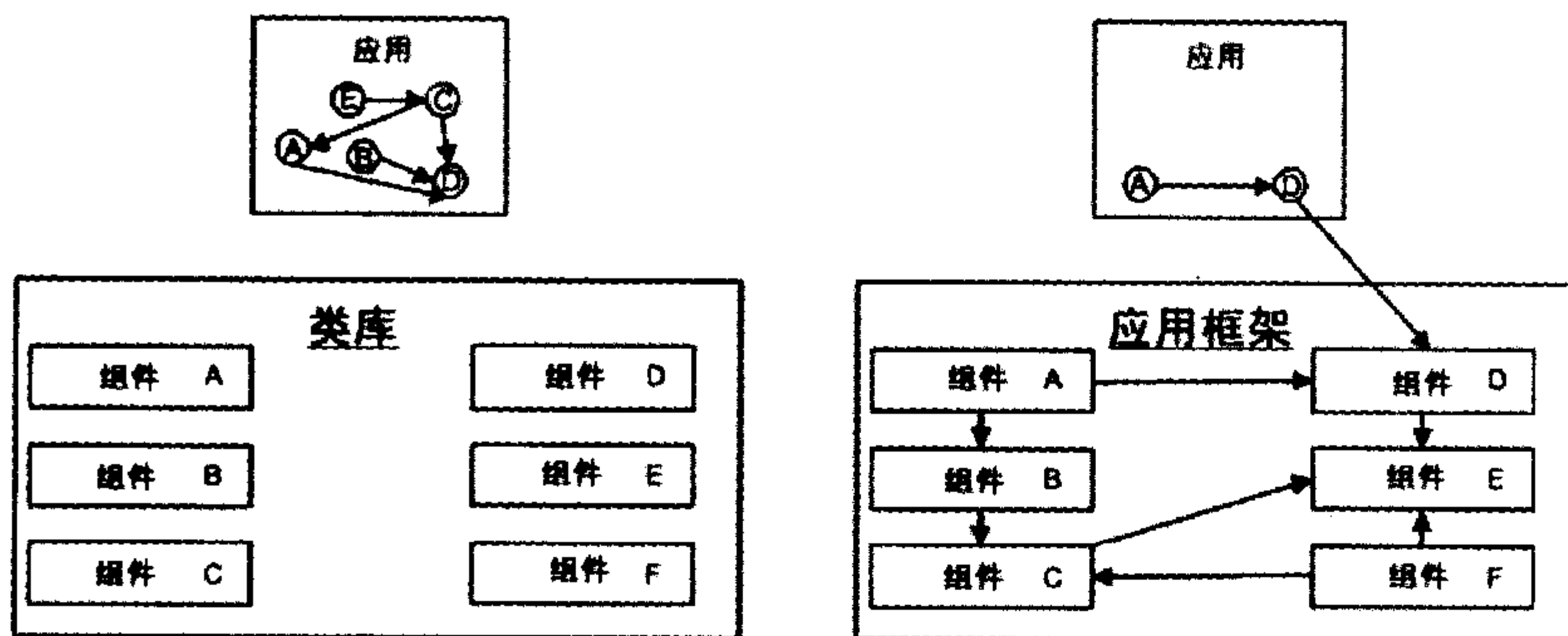


图 2-15 类库与框架（图片来源：《应用框架的设计与实现——.NET 平台》）

Xin Chen 在他的著作《应用框架的设计与实现——.NET 平台》一书中写道：

框架通过封装处理流程的控制逻辑，使它对开发者透明，来简化开发工作。这种封装也是框架和类库（class library）的区别之一。类库由许多现成的、供开发者用于构建应用的组件组成，但是，开发者必须理解不同组件之间的关系，并编写处理流程代码把众多组件组织起来。框架则不同，它通过预先把众多组件组织在一起的方式，封装了处理流程的控制逻辑；因此，开发者就不用再编写控制逻辑来组织组件之间的交互了。……

……应用开发者使用类库这种方法时，必须编写管理类库中不同组件实例（instance）的控制流程。为此，应用开发者必须充分理解每个相关组件，以及组织组件协作所必需的业务逻辑。而使用框架这种方法时，由于大部分处理流程已经被框架管理了起来，所以开发者需要编写的控制代码就非常少。由于应用框架隐藏了不同组件之间的处理流程，这就免去了开发者编写协调逻辑（coordination logic）之苦，也不用经历编写这些协调代码的学习曲线了。既然处理流程的控制逻辑从应用层移到了应用框架层，那么框架的设计人员就要运用其架构和领域知识，来定义框架内的组件该如何协作；而使用框架的开发者，几乎无须知道框架组件如何协作，就能高效地开发应用。

由此可见，从重用的角度来比较，框架提供的重用性比类库更大。类库的目标是提供通用的类，如果是 Utility 类的情况（包含多个 static 方法），将其目标理解成提供通用的函数（就像 C 语言时代的函数库一样）也未尝不可；而框架的目标是提供在某领域内通用的软件系统半成品（或子系统半成品）。

## 2.5.2 框架的分类

接下来，谈谈框架的分类。

框架可以有多种相互独立的分类方式，如图 2-16 所示。





API 进行了封装，使得 MFC 的功能远远超出了一般框架的应用范围；因此，MFC 也带有基础设施框架的性质。

还可以将框架分为：技术框架和业务框架。

- 技术框架致力于解决某一技术领域的通用技术问题，并提供定制和扩展机制。例子很多，例如 Hibernate 就是解决 ORM 问题的技术框架；
- 业务框架则是在特定业务领域内通用的框架，比如 workflow 领域的框架有国内 Huihoo 动力推出的开源 workflow 引擎 Willow (<http://sourceforge.net/projects/huihoo>)，CRM 领域的例子有 SugarCRM 等。

技术框架又称为水平框架，业务框架又称为垂直框架。所谓水平，强调的是通用性、适用范围的广泛性，例如一个线程池框架就可以用于各种项目；所谓垂直，则强调专门化，例如一个网络管理软件的垂直框架针对网络管理这个专门领域提供了完善的功能。

从开发人员的角度，他们最关心的是黑盒框架、白盒框架、灰盒框架的分类法。这种分类主要是根据扩展点的机制来划分的。白盒框架有利于面向对象的继承和多态机制支持扩展点，而黑盒框架更强调接口机制。至于灰盒框架，是实际中应用最多的框架，它兼有白盒框架和黑盒框架的性质，混合使用上述两种机制。

### 2.5.3 框架的开发过程

在 Xin Chen 所著的《应用框架的设计与实现》一书中，介绍了框架的开发过程，非常清楚。

框架的整个开发过程，包括四个主要的阶段，即分析阶段、设计阶段、实现阶段和稳定阶段。如图 2-17 所示，其中灰色的内圈代表阶段，而外圈代表每个阶段的主要任务。

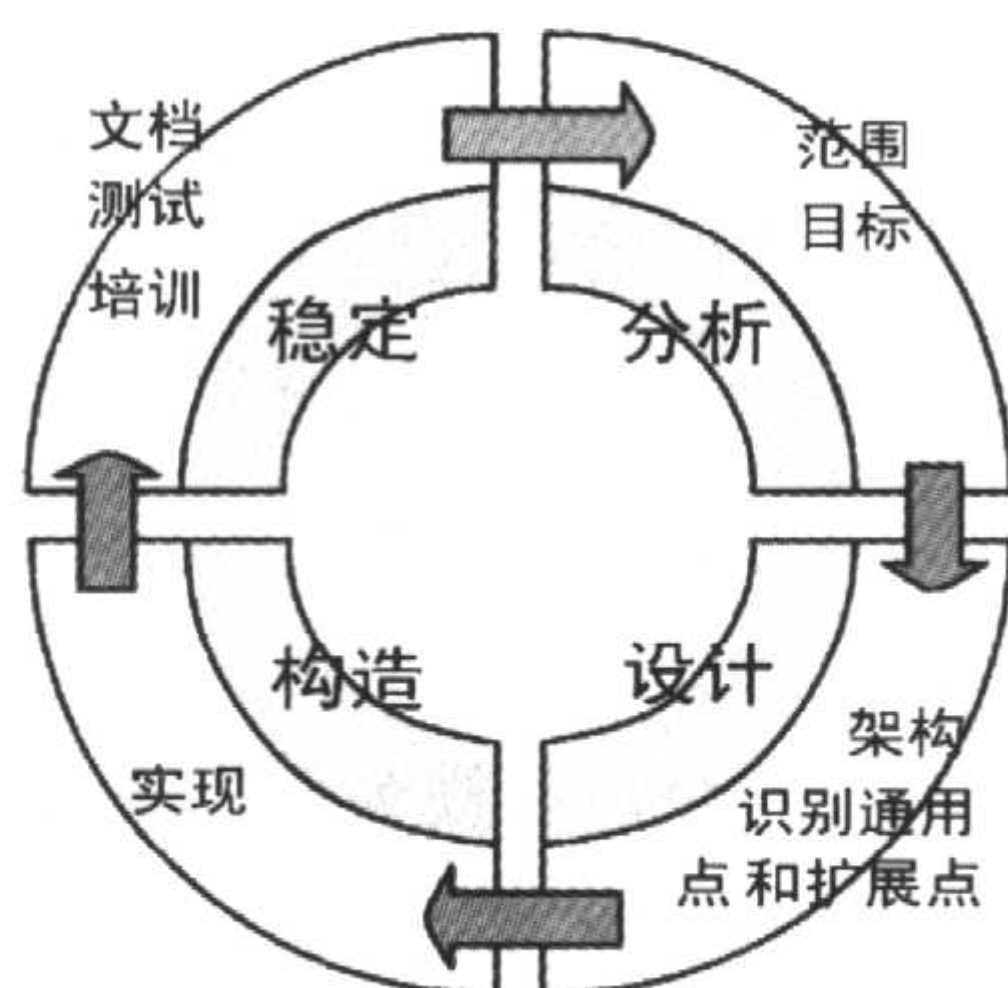


图 2-17 框架开发过程（图片来源《应用框架的设计与实现——.NET 平台》）

《应用框架的设计与实现——.NET 平台》中写道：

要开发应用框架，第一个阶段是分析阶段。和应用开发一样，框架开发首先要确定框架的范围（scope）和目标（objective）。

.....

确定了应用框架的范围和目标之后，下一个阶段是设计阶段。该阶段有两大任务。首先，对特定领域框架层和跨领域框架层都要识别出通用点（common spot）和扩展点（hot spot）。其次，为框架设计架构（architecture），它将用作实现阶段的蓝图。

.....

在业务专家和软件架构师列出了待开发的组件和服务，并区分出了哪些是通用点哪些是扩展点之后，软件架构师就可以开始设计框架的架构了。其间，软件架构师应当提交一些设计级的可提交工作产品，比如类图和活动图，它们在实现阶段将被使用。软件架构师还应考虑与组件和服务设计相关的技术，比如设计模式，以使最终的框架代码具有更好的可重用性和易扩展性。

在设计阶段，也可以创建应用框架原型（prototype），然后在其上构建一个样本应用；通过该样本应用测试了应用框架原型，有助于你了解你所开发的框架如何用于构建业务应用，并洞察到框架设计中潜在的可改进之处。

在框架设计阶段之后，是框架的编码实现阶段。.....

稳定阶段是一个迭代周期的最后一个阶段，它的工作集中在测试、修改 Bug、开发者反馈、写文档和进行培训。

和应用系统的开发过程相比，框架开发在稳定阶段有最大的不同。比如，在谁来测试的问题上就有极大的不同：应用系统一般由专门的测试人员或者最终用户负责进行测试，而框架的测试主要由即将使用该框架的开发人员来测试。

框架开发的稳定阶段还应提供相关的框架文档，和应用系统所提供的文档也有明显差异。正如我们在 Sourceforge 网站上所看到的，框架应提供的文档主要包括：

- 框架介绍；
- Step by step 的入门教程；
- 开发者指南；
- API 速查手册；
- 另外应提供一组例子。

2.5.4 如何实现框架中的扩展点

表 2-1 对如何实现框架中的扩展点进行了总结。

表 2-1 如何实现框架中的扩展点

技术分类	回调机制	开发手段
传统编程语言	函数指针	函数指针作参数
面向对象编程语言	抽象方法（或称虚函数）	抽象基类
		接口
其他技术 （可以和语言无关）	约定数据的格式	提供特定格式的数据

传统编程语言还大有生命力，例如通讯领域、系统软件领域和嵌入式领域等都离不开 C 等传统编程语言。这些编程语言也可以实现框架，虽然没有面向对象框架那么多地被提及，但业界很多人都是这么做的。如图 2-18 所示，过程化编程语言实现的框架，其基本元素是函数（例如 C 语言）；为了实现框架对定制函数的逆向控制，我们让框架通过一个函数指针对定制函数进行“回调（Callback）”。

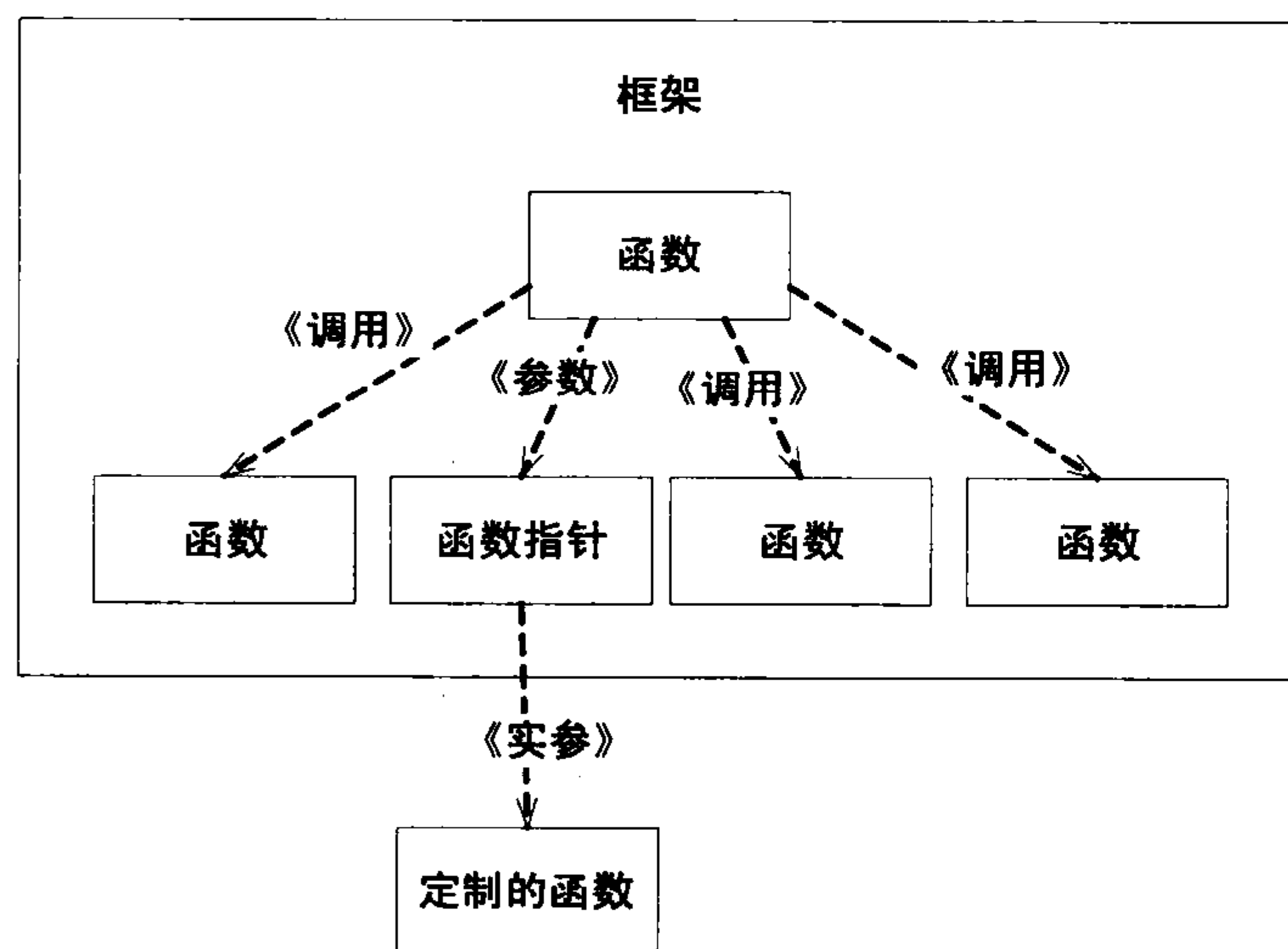


图 2-18 用过程化语言实现框架

举个例子，C 语言的标准库（头文件为 `stdlib.h`）实现的 `qsort()` 用来支持快排序，它的函数原型为：

```

void qsort(
    void *base,
    size_t num,
    size_t width,
    int (__cdecl *compare)(const void *elem1,
        const void *elem2)
);
  
```

其中的第三个名为 `compare` 的形参是一个函数指针，开发者只需自己实现特定的比较函数，



就可以借助 `qsort()` 实现快排序。例如，最简单的当然是对 `int` 类型的数组进行排序：

```
int num[100];

int cmp ( const void * i , const void * j )
{
    return *(int *)i - *(int *)j;
}

qsort(num,100,sizeof(num[0]),cmp);
```

再例如，对一个复杂的“结构”进行排序：

```
struct employee {
    int employee_num;
    char[100] employee_name;
    ...
} employees[100];

int cmp( const void *a , const void *b )
{
    struct employee *c = (employee *)a;
    struct employee *d = (employee *)b;
    return c-> employee_num - d-> employee_num;
}

qsort(employees,100,sizeof(employees [0]),cmp);
```

可以说，面向对象技术的发展极大地提高了框架的能力，并推动了框架技术被普遍接受。如图 2-19 所示，面向对象框架的组成部分包括具体类、抽象类和接口。抽象方法是面向对象支持“多态”的关键，面向对象框架借助抽象方法实现逆向控制。

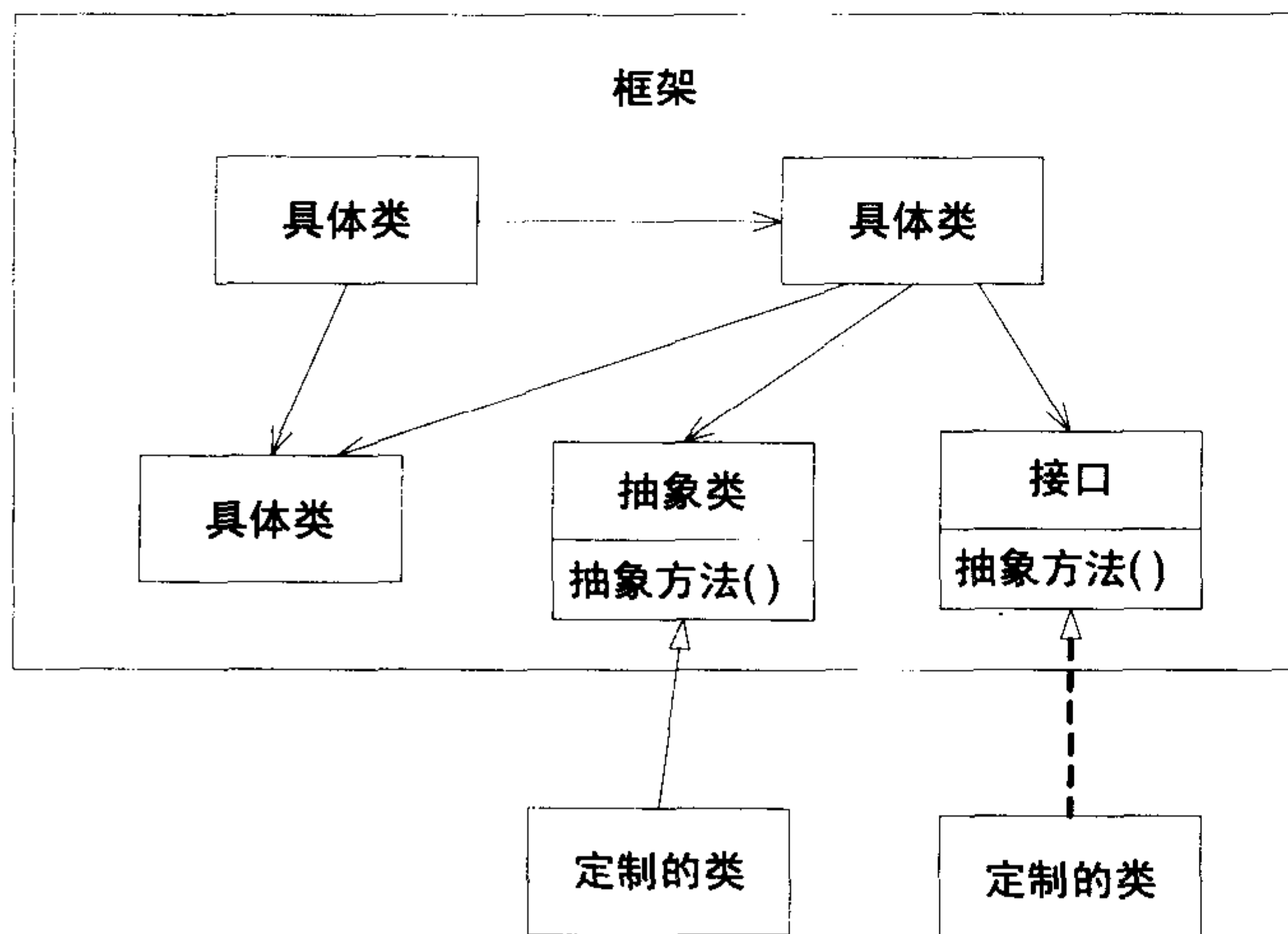


图 2-19 面向对象框架

更广泛而言，还可以通过“数据驱动”来实现控制反转。具体来说，数据驱动大致可以分为

两类：配置驱动和元模型驱动。如图 2-20 所示，框架按照约定的数据格式读取用户提供的数  
据，这些数据将决定一些未确定行为的具体执行。

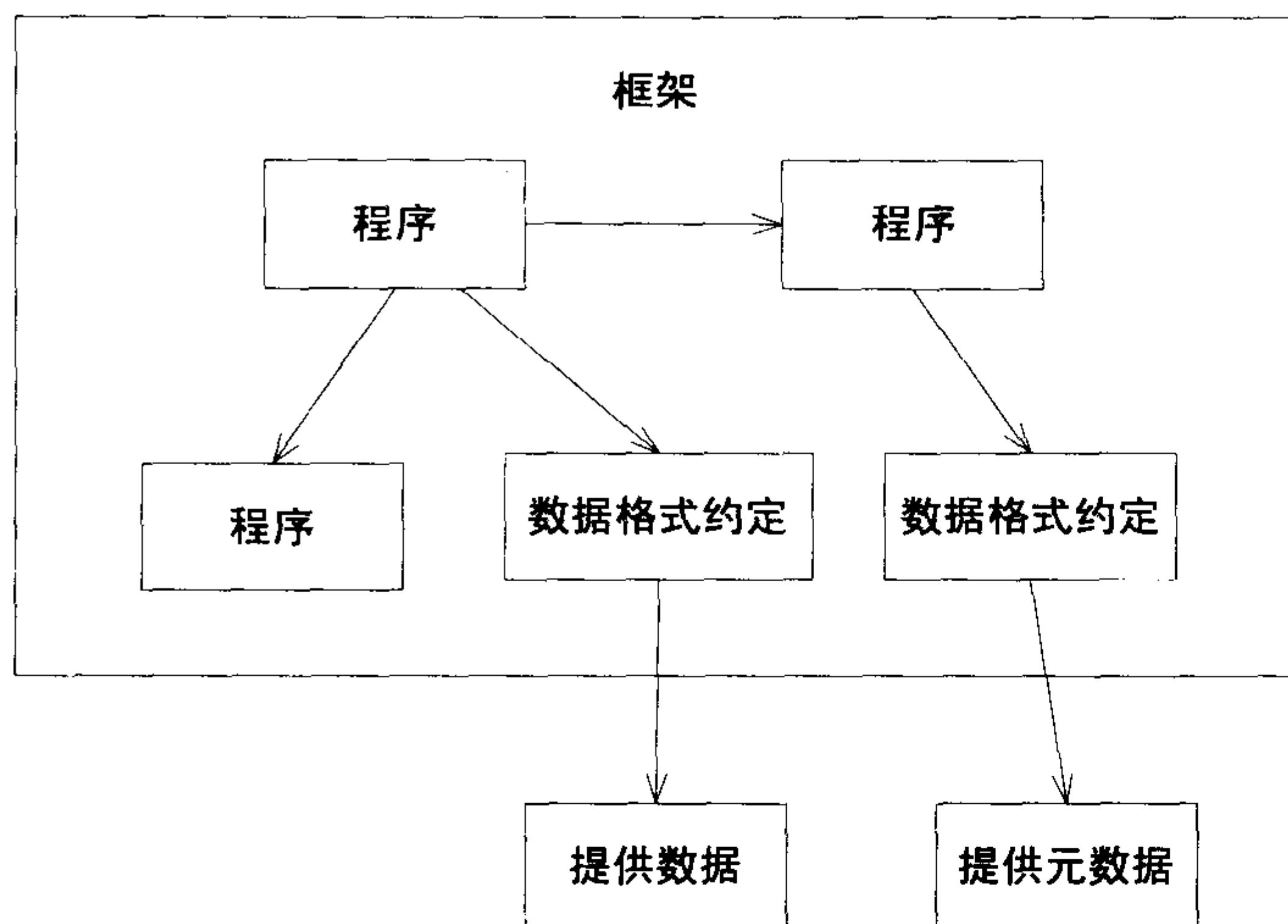


图 2-20 数据驱动的框架

许多面向对象框架在利用抽象方法支持扩展的同时，还会借助“配置驱动”来降低使用框架的难度。例如，Struts 如果不利用 struts-config.xml 配置文件的话，可以把 ActionServlet 改为抽象类，要求开发者实现它的子类；显然，这种方式太麻烦了。总之，框架中的总控类采用配置驱动往往比编写其子类更为方便。

## 2.6 总结与强调

在软件架构实践中，子系统、框架与架构的关系是无法回避的问题，是软件架构师的“必修课”。

本章揭示了软件系统是由不同粒度的软件单元层层递归构成的这一事实，而我们在实践中又必须注意，同一个软件单元在不同实践者眼中的粒度可能不同。只有懂得了这一点，软件架构师才能够游刃有余地根据情况忽略应该被忽略的细节，抓住设计大局。软件子系统也可能需要架构设计；而即使是同一系统内部，子系统不同，所采用的架构也有可能不同。这在实践中屡见不鲜。

框架和架构既有区别又有联系。框架也需要架构设计；反过来，可以通过架构框架化达到“架构重用”的目的。

本章也给出了软件架构的概念模型，它更深入地揭示了软件架构的本质。

最后的框架技术专题区分了类库与框架、说明了框架的分类、讨论了框架的开发过程和开发技术。

## 第3章 软件架构的作用

---

在没有架构的情况下，建造任何高复杂度的结构都是鲁莽的。

——Ivar Jacobson, 《统一软件开发过程之路》

如果一个项目的系统架构（包括理论基础）尚未确定，就不应该进行此系统的全面开发。

——Barry Boehm, 《Engineering Context》

错误守恒定律指出，在一个大型系统内仍然存在的错误数目和已经修正的错误数目成正比。换句话说，一个缺陷充斥的系统将始终是一个缺陷充斥的系统。这是因为一个设计拙劣的系统不仅仅开始时有更多的缺陷，而且它们也难以正确修复，因此会更严重地受到波纹效应的影响。

——Timothy C. Lethbridge, 《面向对象软件工程》

你既然已在阅读本书，就意味着你了解软件架构的重要性。本章主要讨论软件架构在不同实践场景下的作用，掌握软件架构在实际工作中的意义。

### 3.1 充分发挥软件架构的作用

---

软件架构是软件开发过程初期的产物，这一点是没有异议的。但是，为了充分发挥软件架构的作用，应进一步认识到以下两点：

- 软件架构对后期的软件维护，乃至改动力度比较大的软件升级都起着重要作用；
- 越来越多的公司和企业开始注重产品线的开发，这时需要为整个产品线设计软件架构，这和为单个产品设计软件架构有很大不同。

图 3-1 展示了软件架构在 4 种不同情况下所起的不同作用，并重点说明了它们在时间轴上的差异以及它们在工作难度上的各不相同。



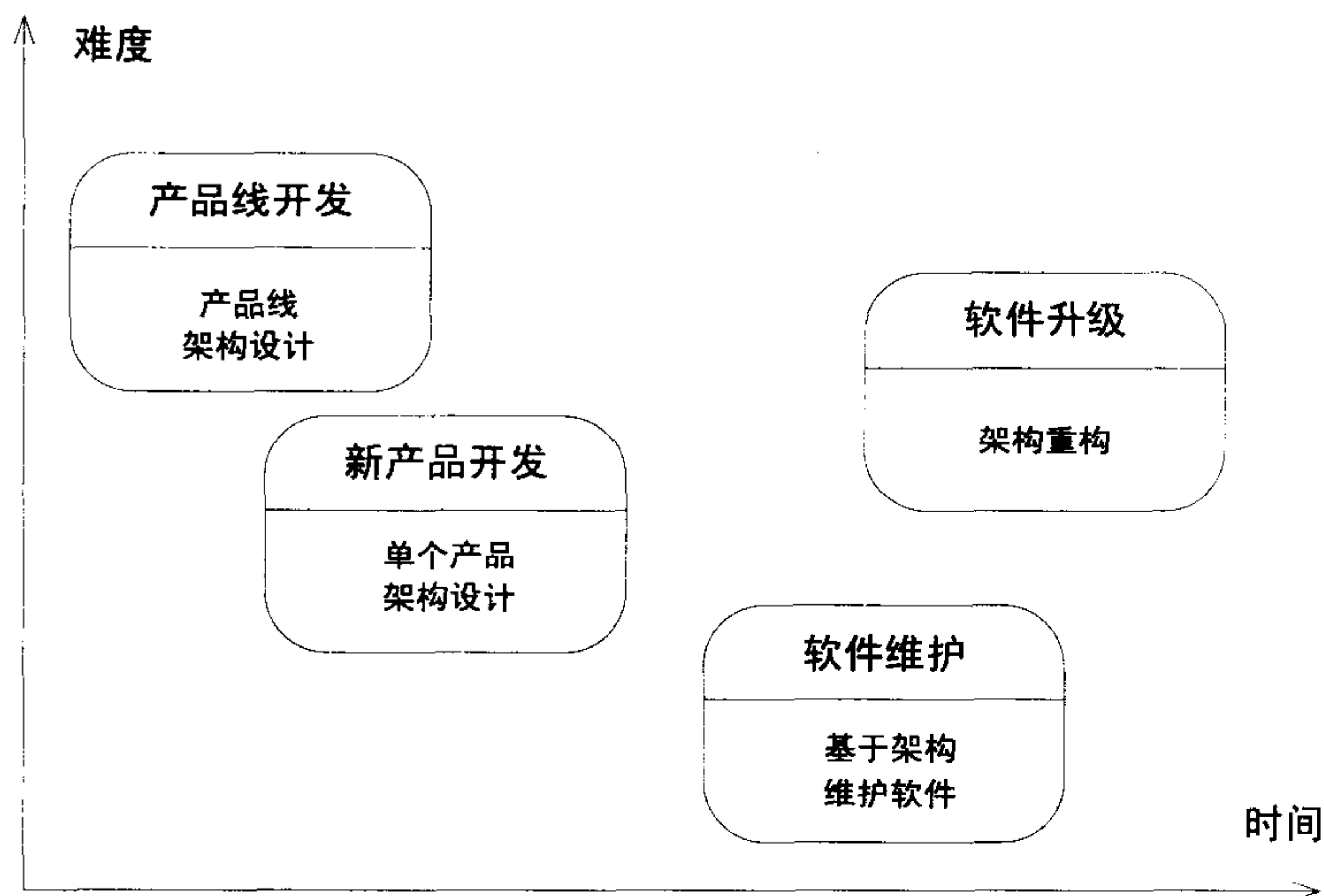


图 3-1 软件架构的作用

从时间上来讲，产品线架构设计和单个产品架构设计发生在过程早期，而基于架构的软件维护和架构重构则发生在过程后期。如果有产品线架构的话，单个产品架构设计要在产品线架构设计之后完成，因为此时每个产品的设计都应在遵循整个产品线架构设计的基础上引入个性化的设计。

当软件系统交付上线之后，还需要不断地对它进行维护，其中既包括对错误的修改性维护，也包括对新需求的改善性维护。

对软件系统不断的修改会使系统架构慢慢变得混乱，于是有一天，当我们碰到下列两种情况中的一种时，我们下决心要将架构进行重构以使它更加合理清晰：

- 架构太混乱了，以致进行一个小修改都会牵动全身；
- 将要进行的软件升级力度很大，原先的架构已不再适应新需求了。

于是这时，我们就会对软件架构进行比较大的修改和调整，使它适应新需求以及开发和维护的需要。

从难度上来讲，产品线架构设计是最难的，因为它要考虑更多企业发展因素、引入更多对变化的支持。比较而言，基于架构的软件维护相对容易些。本书认为，架构重构有时并不比新产品的架构设计容易。这是因为：你不仅要全面把握需求，还要充分理解原有架构的利与弊，把合理的设计保留下来——如果产品已经上线，则架构设计面临的限制就更多了。

## 3.2 软件架构对新产品开发的作用

软件架构设计为什么这么难？因为它是跨越现实世界与计算机世界之间鸿沟的一座桥。

从面向业务的需求，到最终的面向技术的软件系统，要跨越很大的鸿沟。软件架构设计就是要完成从面向业务到面向技术的转换，在鸿沟上架起一座桥梁。软件架构师根据各种需求进行架构设计，最终的软件架构包含了结构、协作和技术等方面的重要决策，为系统化的开发活动建立了基础（如图 3-2 所示）。

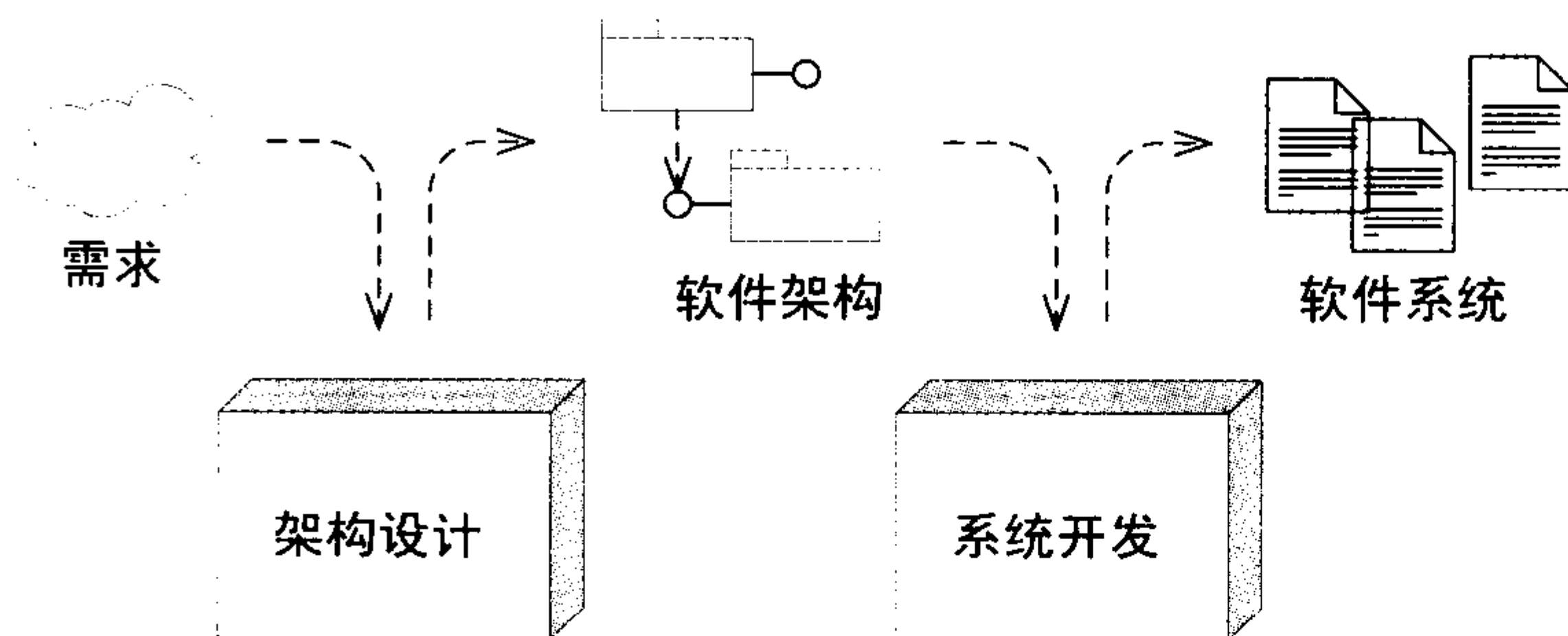


图 3-2 软件架构的位置

具体而言，软件架构对新产品开发的作用包括以下几个方面。

**上承业务目标。**不能为用户和客户实现特定目标的软件系统是没有生命力的，而软件架构设计（和其上游的需求活动一起）担负着为完成业务目标而进行大局规划的职责。软件架构的设计必须满足用户或客户的业务目标（如加速周转）、业务功能（如个人理财服务）、约束限制（如下一个财年必须上线）和质量属性（如高持续可用性）等方面的需求。

**下接技术决策。**软件开发最终生产出可运行的软件系统，它是面向技术的。软件架构师将面向业务的需求转化为面向技术的软件架构设计方案，为后面的技术开发工作提供了切实的指导和限制。

**控制复杂性。**一股脑地将复杂性统统展开，就会使软件开发陷入混乱和无法控制之中。先进行架构设计，后进行详细设计和编码实现，运用了“基于问题深度分而治之”的理念，利于控制复杂性。正因为如此，更多的人能够理解将要开发的软件系统，使开发人员、管理人员、用户和客户等都理解他们将要参与的工作，并成为他们开展交流的基础。

**组织开发。**软件架构为开展系统化的团队开发奠定了基础，它规定了软件系统的各元素如何彼此相关的设计决策，从而可以把不同模块分配给不同小组分头并行开发，而软件架构设计方案在这些小组中间扮演了“桥梁”和“合作契约”的作用。每个小组的工作覆盖了“整个问题的一部分”，各小组之间可以互相独立地进行并行工作。

**利于迭代开发和增量交付。**早交付、多反馈，是提高客户满意度的最佳实践之一。以架构为中心来进行软件开发，为增量交付提供了良好的基础。这是因为性能、可测试性和可扩展性等大多数非功能需求更大程度上依赖于软件架构的设计（而非编程实现），所以在软件架构设计方案经过验证之后，我们就可以专注于功能的增量提交，进行迭代式的软件开发。

**提高质量。**这一点可以认为是“控制复杂性”和“利于增量开发”的衍生优点。清晰的软件架构将各个模块的职责划分得有条不紊，每个模块都有清晰的接口，这相当于间接降低了开发难度（或者说，混乱的架构人为地增加了开发难度），利于提高软件质量。另一方面，以架构为中心的增量开发会不断地发布软件系统的可执行版本，最初的版本所包含的业务功能并不多，这时可以重点测试软件架构对质量属性需求的满足程度，并及早作出架构调整。这显然也利于提高软件质量。

### 3.3 软件架构对软件产品线开发的作用

什么是软件产品线？

《软件产品线实践与模式》指出：“软件产品线是指具有一组可管理的、公共特性的、软件密集性系统的集合，这些系统满足特定的市场需求或任务需求，并且按预定义的方式从一个公共的核心资产集开发得到。”

什么是软件产品线架构？

所谓软件产品线架构，是针对一个公司或组织内的一系列产品而设计的通用架构。这一系列产品必须有很多相似性，从而它们可以共享同一个架构和部分具体实现，提高生产率。

在实践中，软件产品线架构除了定义组成产品的各部分的职责，以及它们之间的交互之外，还往往：

- 将系列产品共用的模块事先实现，供直接重用；
- 有时还将架构设计方案用框架的形式予以实现，供定制使用。

此时，软件产品线架构就有了一个流行的名字：平台。

产品线架构和单个产品架构的区别何在？

《软件产品线实践与模式》归纳了三点产品线架构的“特有之处”：

- 产品线架构必须考虑一系列明确许可的变化（如图 3-3 所示）；
- 产品线架构一定要文档化；
- 必须提供“产品创建者指南”，描述架构的实例化过程。



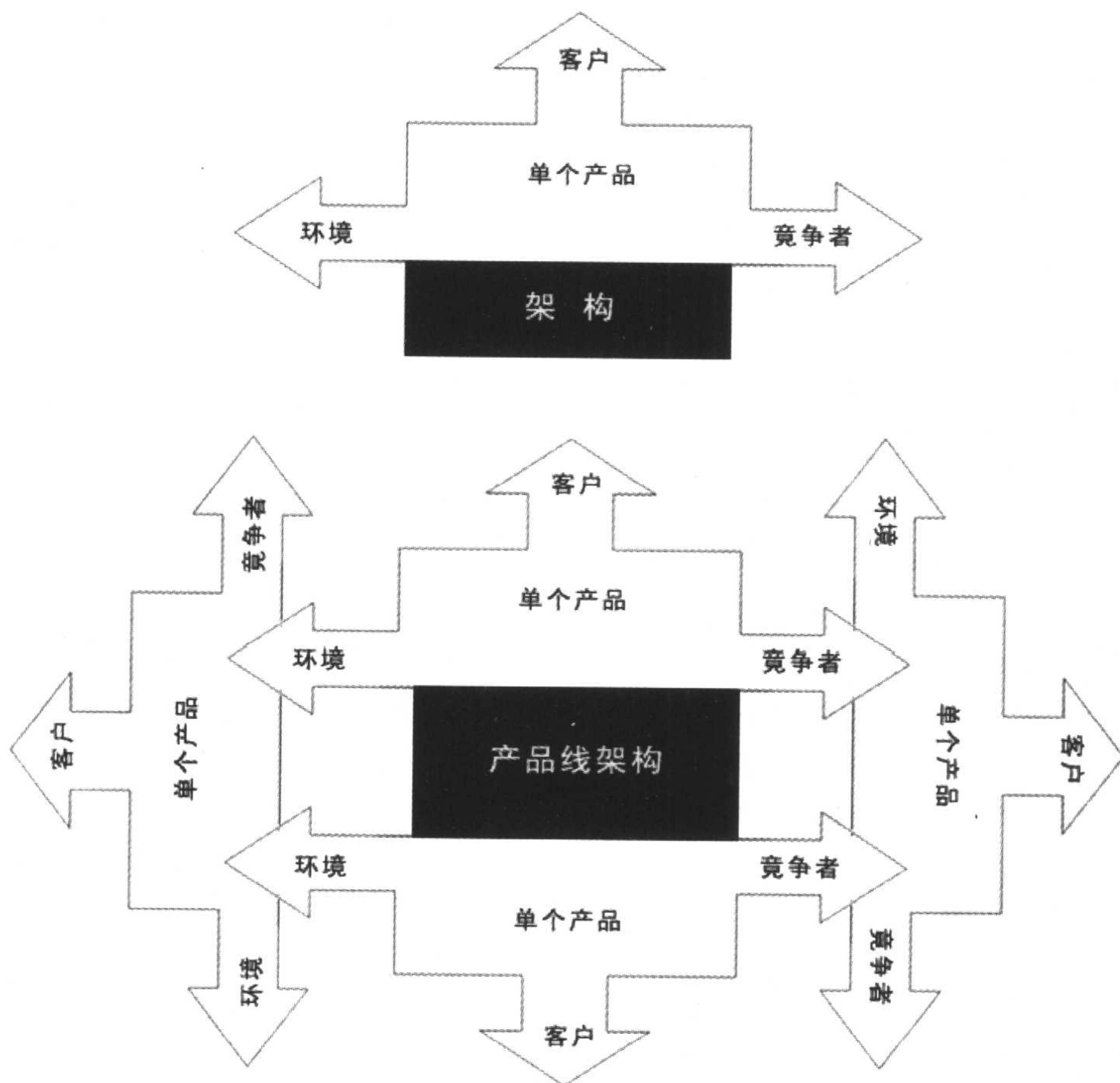


图 3-3 产品线架构必须预见更多变化（图片来源：《软件架构：组织原则与模式》）

构成软件产品线的不同产品可能是针对不同的客户、竞争者或使用环境推出的，这意味着它们是同时存在的，但它们在功能、质量属性、目标平台、所用中间件和规模等方面都可能有所不同。例如，某进销存软件的专业版使用 Access 数据库，而它的企业版要求必须使用 Oracle 数据库。总之，产品线架构必须考虑一系列明确许可的变化。

我一直在想“源代码就是设计”是否适用于软件产品线的情况。你可能是个敏捷方法的实践者，如果你在进行项目或单个产品的架构设计时没有写《架构设计文档》，你的同事可以读代码甚至通过重构来理解你的设计，但对软件产品线来说可不是“代码集体所有”……所以，产品线架构一定要文档化是有道理的。

了解了上述知识，我们就不难理解产品线架构对软件产品线开发的作用：

- 固化核心知识;
- 提供可重用资产;
- 缩短推出产品的周期;
- 降低开发和维护总成本;
- 提高产品质量;
- 支持批量定制。

### 3.4 软件架构对软件维护的作用

软件系统交付上线了,就需要不断地对它进行维护,其中既包括对错误进行修改的“修改性维护”,也包括增加新功能的“增强性维护”。

如图 3-4 所示,运行中的软件系统会暴露出一些 Bug 需要修改,而用户也可能提出一些新功能的要求,这是维护工作的两种典型来源。而软件架构是软件维护的基础,一个 Bug 的修复或一个新功能的增加,往往涉及架构中的一条“模块协作链”,因此了解架构将有利于维护工作;相反,不了解架构而盲目地修改程序可能违背架构设计的思路,使整个系统的架构慢慢变得混乱,并可能引发出其他意想不到的 Bug 和问题。

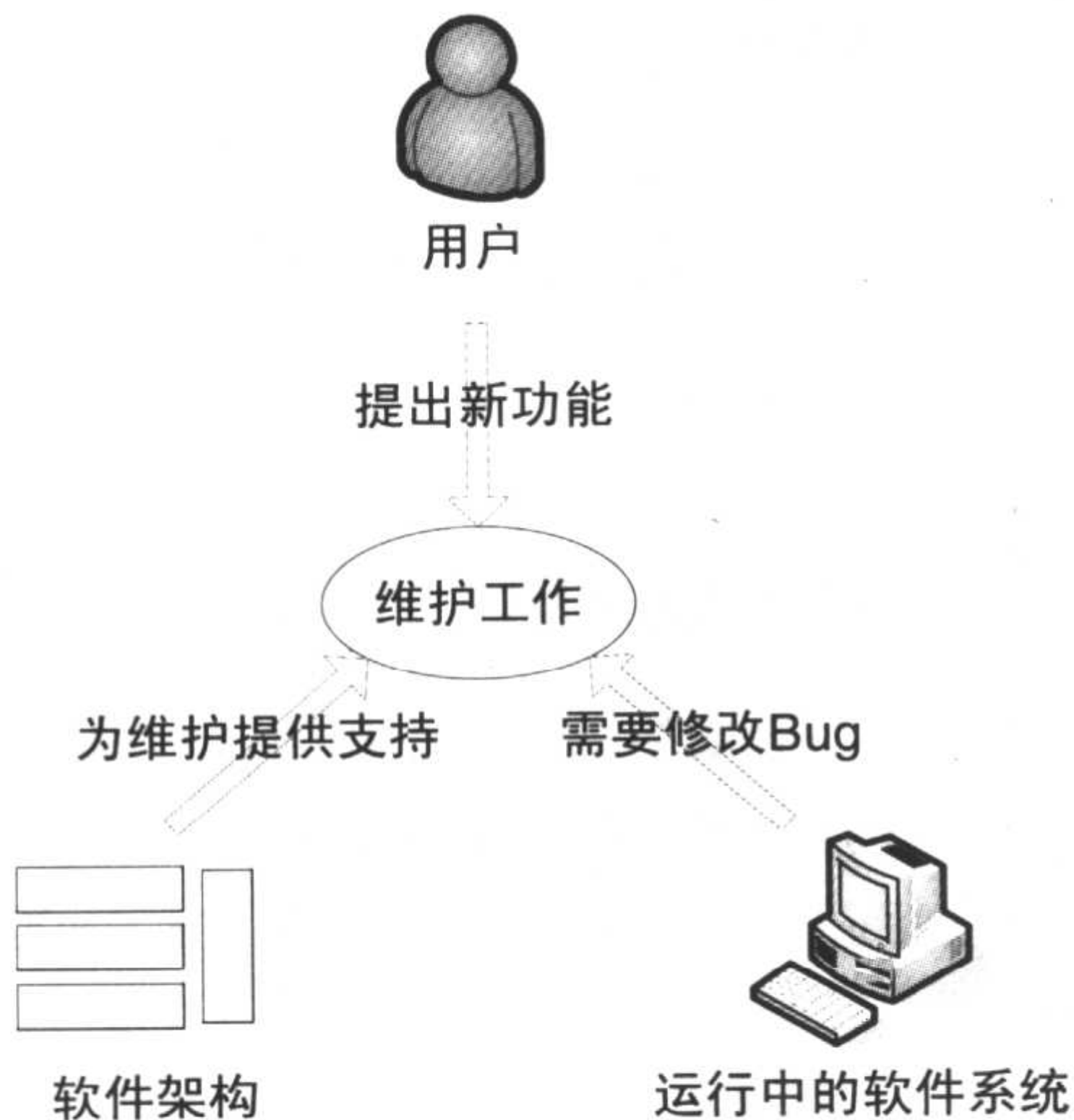


图 3-4 软件维护

### 3.5 软件架构重构

软件架构会“磨损”——随着对软件系统不断地修改,软件架构将变得混乱。

直到有一天，当我们碰到下列两种情况中的一种时，我们会下决心进行架构重构：

- 架构太混乱了，以致进行一个小修改都会牵动全身；
- 将要进行的软件升级力度很大，原先的架构已不再适应新需求了。

所谓软件架构重构，是指对软件架构进行比较大的修改和调整，使它适应新需求及开发和维护的需要。软件架构重构属于“再工程（Reengineering）”的一种情况，一般会经历逆向工程、重新规划、正向工程 3 个步骤（如图 3-5 所示）。

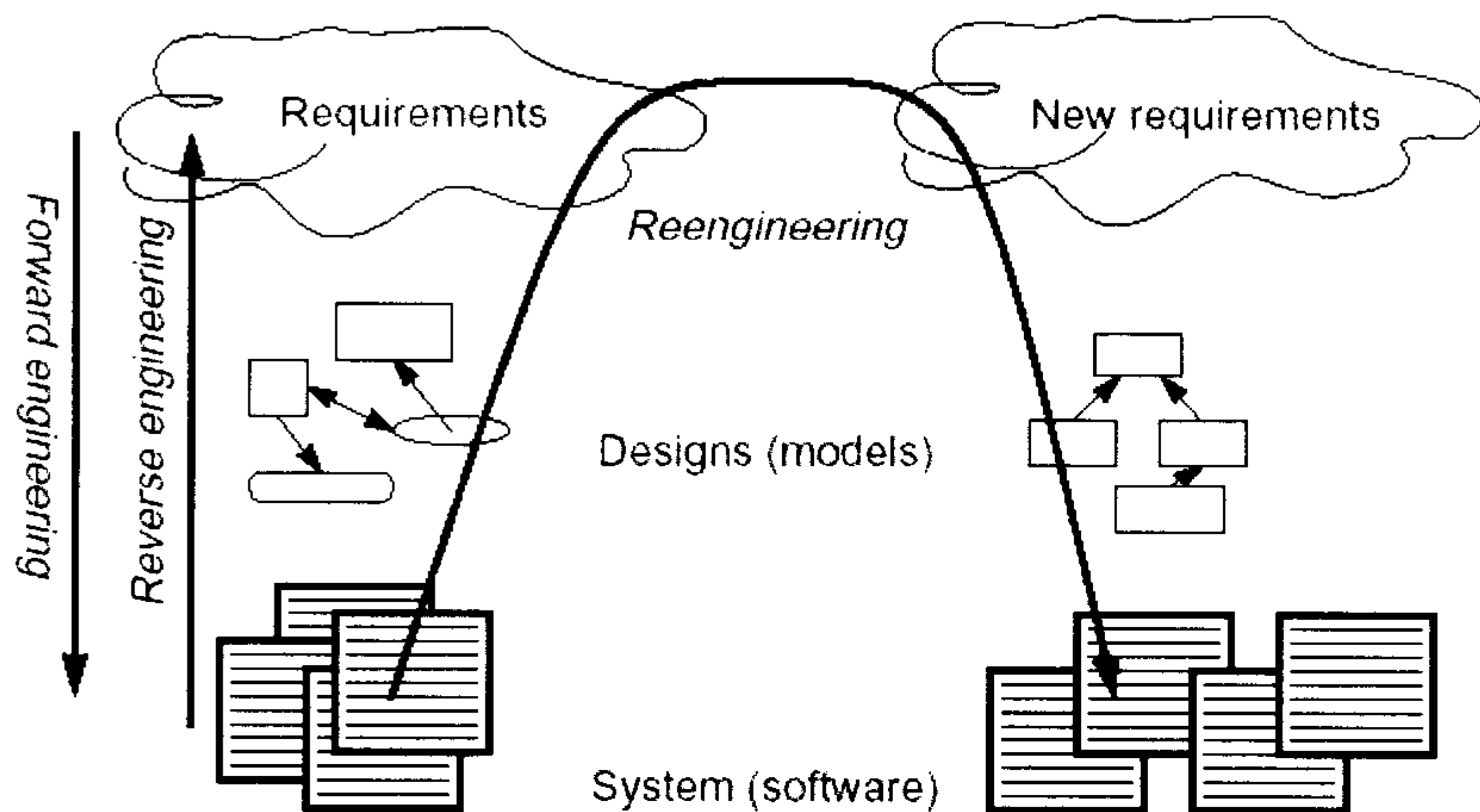


图 3-5 再工程的马蹄铁模型（图片来源：《面向对象的软件再工程模式》）

其实，逆向工程并不一定总意味着“从二进制到源程序”，根据源程序、现有文档或知情人访谈等途径了解系统的设计乃至需求也属于逆向工程。只有充分理解原有架构的设计思路，才能评估它在现有需求情况下的“利”与“弊”，把合理的设计保留下来，把不妥的设计修改掉。

## 3.6 总结与强调

很多人不了解如何在不同情况下充分发挥软件架构的作用，本章就是特意针对这种现实而写的。我们应该分门别类地掌握软件架构在下列工作中的意义：

- 新产品开发；
- 软件产品线开发；
- 软件维护；
- 软件升级。

本书提供了丰富的实战案例（本书目录之前有“案例-章节对照表”），其中，第 6 章和第 8 章的“网络管理产品线案例”是专门针对产品线架构设计的，请读者参考。



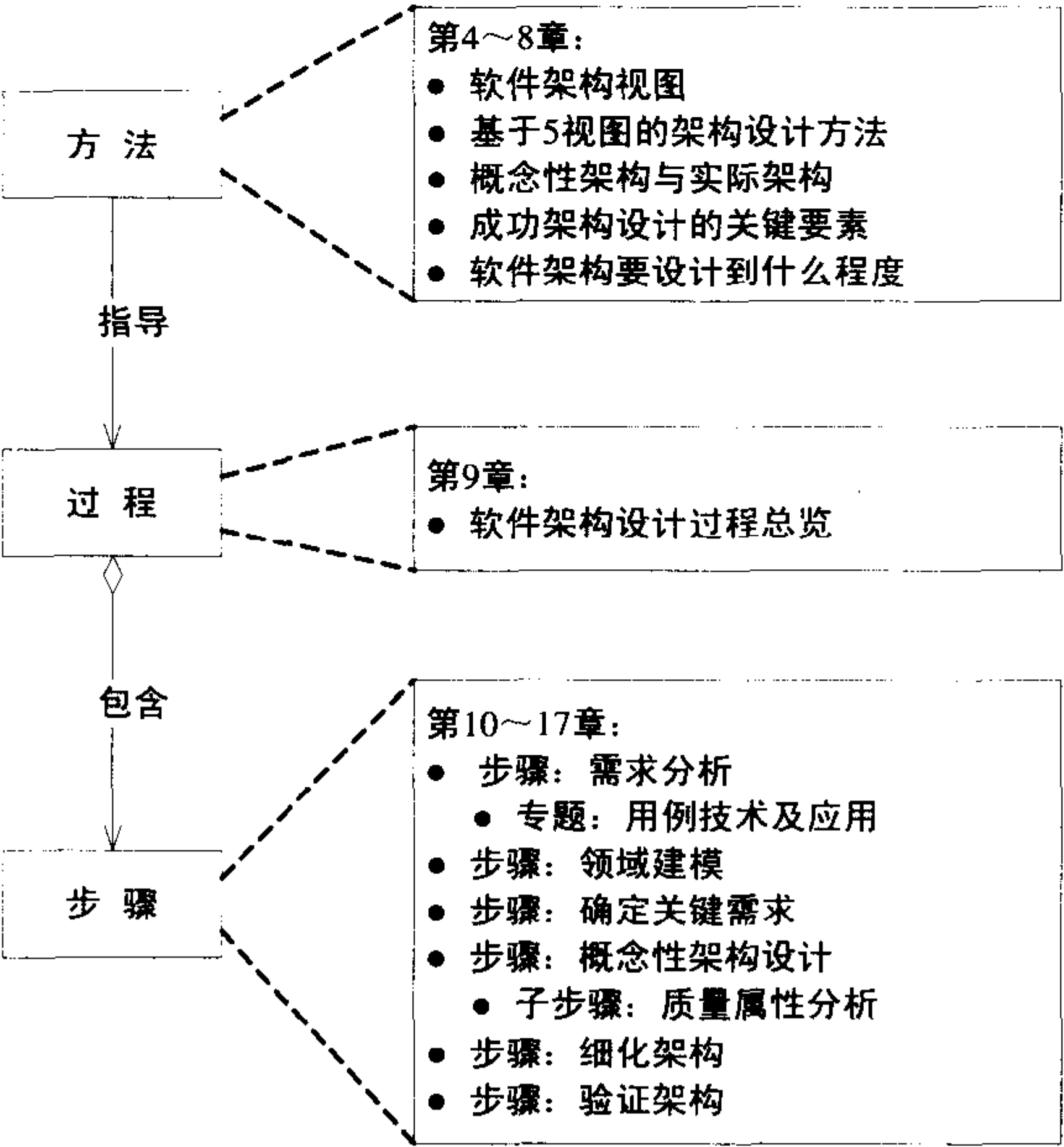


# 第二部分

## 软件架构设计方法与过程篇

第二部分的主题是软件架构设计的方法与过程，共包含 14 章。

方法指导过程，过程包含步骤。这就是第二部分的组织方式。







## 第4章 软件架构视图

---

横看成岭侧成峰，远近高低各不同。

——苏轼，《题西林壁》

软件架构是一种无法以简单的一维方式进行说明的复杂实体。

——Paul Clements, 《软件架构编档》

多重软件架构视图之所以必不可少，是因为各类涉众（用户、客户、开发人员、测试人员、维护人员、内部操作人员、其他人员）需要从各自的角度理解和使用架构。

——Barry Boehm

通过第一部分的介绍，我们了解到，所谓软件架构就是关于如何构建软件的一些最重要的设计的决策，这些决策往往是围绕将系统分为哪些部分、各部分之间如何交互展开的。

然而，为了开展软件架构设计，还应引入一些新的方法和思路，其中“软件架构视图”就是架构设计方法中非常关键的一个概念。

鉴于此，本章将讨论“软件架构视图”这一主题：

- 首先，说明在没有软件架构视图概念的情况下，实践中所面临的一些问题；
- 之后，讨论软件架构师到底为谁而设计的话题，说明软件架构师必须充分考虑来自用户、客户、开发人员和管理人员的需求和限制；
- 然后，阐述软件架构视图的概念，并讨论逻辑架构和物理架构——这是最为常见的两种软件架构视图；
- 最后，通过一个实际案例说明逻辑架构和物理架构是如何指导架构设计的。

### 4.1 呼唤软件架构视图

---

在现代的软件开发中，软件架构是进行团队开发的基础，因此兼顾不同角色视角的多重架构

视图方法是必不可少的。

### 4.1.1 办公室里的争论

办公室里，关于什么是软件架构，争论正酣。

程序员说，软件架构就是要决定需要编写哪些类、使用哪些现成框架（Framework）。程序经理笑了；

程序经理说，软件架构就是模块的划分和接口的定义。系统分析员笑了；

系统分析员说，软件架构就是为业务领域对象的关系建模。配置管理员笑了；

配置管理员说，软件架构就是开发出来的以及编译过后的软件到底是个啥结构。数据库工程师笑了；

数据库工程师说，软件架构规定了持久化数据的结构，其他一切只不过是对数据的操作而已。部署工程师笑了；

部署工程师说，软件架构规定了软件部署到硬件的策略。用户笑了；

用户说，软件架构就是决定一个个功能子系统如何划分。程序员又笑了；

大家想了想说，这些架构视图好像我们都需要啊，软件架构师哭了。

.....

上述争论可以总结为一句话：不同涉众看待软件架构的视角是不同的。

### 4.1.2 呼唤软件架构视图

和任何复杂的事物一样，随着我们对软件架构概念了解的慢慢加深，会有一些比较深入的问题浮现出来。比如，在了解了架构关注“系统分为哪些部分”之后，我们会产生这样一些疑问：

- 从用户角度而言，组成系统的是各种功能的模块，这属于架构设计的范围吗？（属于）
- 对开发人员来说，他们认为系统是由不同的程序包组成的，架构设计师应不应该把这些统统丢给程序员决定呢？（不应该）
- 更进一步而言，运行时系统又是由进程、线程等组成的，这属不属于架构设计的范围呢？（当然属于）

而在了解了架构强调“各部分之间如何交互”之后，我们又会问自己这样一些问题：

- 在用户看来，抽象的功能模块之间可以相互（直接或间接）调用功能服务，只有这样才能完成最终系统需要的业务功能，这是否属于架构设计的决策范围呢？（属于）
- 程序类组成程序包，程序包组成程序系统，这些程序代码之间的调用等交互关系既有局部于包内的，也有跨包进行的，那么，哪些属于架构师应该考虑的呢？（一般而言，某



种级别的程序包之间的交互属于架构设计的范围，这通常会采用定义接口的方式进行。不过，对于习惯把“接口属于客户”原则贯彻到代码结构设计中的架构师而言，以及在进行框架开发的情况下，都有可能出现接口定义在特定包比较密集的情况。)

- 架构可以不关心进程或线程间的通讯和并发等问题吗？毕竟软件系统的性能和可伸缩性等问题与此息息相关啊。(应关心)

由此看来，由于软件架构概念是高度抽象的，所以在软件架构概念与实践之间，似乎存在某种“鸿沟”——即缺失某种概念，而这种概念可以“连接”软件架构的概念和实际的开发实践的需要，为不同涉众理解和交流架构提供更专一的视角。这个概念就是本章要讲的主题：软件架构视图。

## 4.2 软件架构为谁而设计

为了在软件架构纯概念和实践之间搭起一座桥梁，我们将引入**软件架构视图**的概念。在此之前，搞清楚“软件架构为谁而设计”的问题是大有裨益的。

### 4.2.1 为用户而设计

为什么要开发某个软件系统呢？因为要给用户使用：或辅助用户完成日常工作，或帮助用户管理某些信息，或给用户带来娱乐体验……不一而足。

用户要功能，用户也要质量。

每套软件都会提供这样或那样的功能，正是这些功能帮助用户实现他们在工作或生活中的特定目标。用户所需的功能和系统本身的结构一定是相互影响的，这正是软件架构师要特别关注的。先举个生活中的例子吧。例如，因为功能不尽相同，所以“转笔刀”的结构和“专用刀片”的结构也不一样（如图 4-1 所示）：小学生需要削铅笔，那一定是转笔刀最为适合他们，因为方便易用，一转即可；而美术师需要用铅笔来画素描，他们则需要使用专用刀片，因为转笔刀难以削出满足不同绘画要求的形状各异的笔尖。同样，对于软件系统而言，用户需求是千差万别的，我们采用的软件架构必须和所要提供的功能相适应。可以这么说，软件架构师必须时时牢记：为用户而设计。

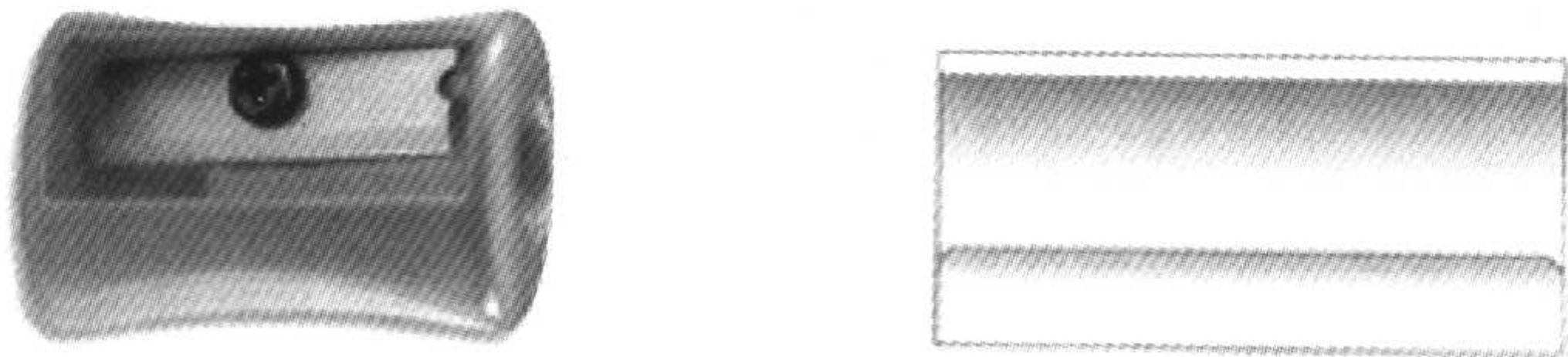


图 4-1 生活中的例子：功能与结构相互影响

诸如性能、易用性等软件质量属性，并不像上面所述的软件功能那样直接帮助用户达到特定



目标，但并不意味着软件质量属性不是必需的——恰恰相反，质量属性差的软件系统大多都不会成功。例如，你提供了用户要求的“交易查询”功能，但这一功能动辄就需要花上几分钟，用户能接受吗？当然不能接受。用户会说，功能虽然具备了，但质量太差难以接受。用户在使用软件系统的过程中，其关心的质量属性可能包括易用性、性能、可伸缩性、持续可用性和鲁棒性等。因此，软件架构师也应当时时牢记：为用户而设计，不仅满足用户要求的功能，也要达到用户期望的质量。

### 4.2.2 为客户而设计

有时，客户不一定是最终用户。例如，对超市销售系统而言，客户是某家连锁超市（的老板），而用户则是超市收银员和上货员。

架构师必须为客户而设计：充分考虑客户的业务目标、上线时间的要求、预算限制，以及集成需要等，还要特别关注客户所在领域的业务规则和业务限制。

架构师应当直接或（通过系统分析员）间接地了解 and 掌握上述需求及约束，并深刻理解它们对架构的影响，只有这样才能设计出合适的软件架构。合适的才是最好的，例如，如果客户是一家小型超市，软件和硬件采购的预算都有限，那么你不宜采用依赖太多昂贵中间件的软件架构设计方案。

### 4.2.3 为开发人员而设计

不知是否是受瀑布式过程的影响太深所致，软件架构师研究需求，设计出软件架构，然后交由开发人员开发出软件……这是太多人早已习惯了的思维模式。于是大家非常重视架构师要为用户而设计，要为需求而设计，却没有提前为按照架构进行开发的开发人员考虑。这就是思维定式造成负面影响的一个典型例子。

架构师也应为开发人员而设计。

例如，作为软件架构师，你是否想过软件系统的“性能”和“可扩展性”有何“深层次的不同”呢？我们一起来思考：

- 它们都是软件的非功能需求？是的，但我们需要了解它们的不同；
- 性能比可扩展性重要？因为性能不达标用户就会拒绝接受，而可扩展性似乎不是“硬指标”？不能这么说，因为不同的系统有不同的要求，很多软件系统就把可扩展性定为必须满足的“硬指标”；
- 性能是软件运行期质量属性，最关心性能的人其实是客户；而可扩展性是软件开发期质量属性，项目开发人员和负责升级维护的开发人员是最关心可扩展性的人吗？是的，这正是“性能”和“可扩展性”关键的不同之处。

我们之所以要探索所谓的“深层次的不同”，就是为了说明极为重要的一点：其实，并不是

所有需求都来自用户，譬如可扩展性这一开发期质量属性就是开发者的需要。

乘胜追击。软件的可扩展性、可重用性、可移植性、易理解性和易测试性等非功能需求，都属于“软件开发期质量属性”之列，它们都将深刻影响开发人员的工作，使开发更顺畅抑或更艰难。因此，软件架构师必须牢记：**关注“软件运行期质量属性”，为开发人员而设计。**

本书第9章将针对不同种类的需求如何影响架构进行专门讨论，请参阅。

#### 4.2.4 为管理人员而设计

可以这么说，因为软件变得越来越复杂了，所以单兵作战不再是普遍的软件开发方式，取而代之的是团队开发。而团队开发又反过来使软件开发更加复杂，因为现在不仅仅要面临技术复杂性的问题了，还有管理复杂性的问题。

可以这么说，要理清并管理不同开发人员之间的协同工作关系，就应该搞清楚开发人员各自负责的软件模块之间的关系——而后者正是软件架构的使命，这使得软件架构自然而然地成为开发管理的基础。

看来，软件架构师也应为管理人员而设计。

例如，对软件项目管理而言，软件架构应当起到应有的作用：为项目经理制定项目计划、管理项目分工和考核项目进度等提供依据。一方面，软件架构从大局着手，就技术方面的重大问题作出决策，构造一个由粗粒度模块组成的解决方案，从而可以把不同模块分配给不同小组分头开发。另一方面，软件架构设计方案规定了各模块之间如何交互的机制和接口，在开发小组之间起到“沟通桥梁”和“合作契约”的作用。

再例如，对软件配置管理而言，软件架构师亦应顾及。一般而言，在软件架构确定之前，软件配置管理是无法全面开展的。配置管理员应该能够从软件架构方案中了解到开发出来的软件以什么样的目录结构存在，以及编译过后的软件目标模块放到哪个目录等决定，并以此作为制定配置管理基本方案的基础。

#### 4.2.5 总结

如此看来，架构师应当为项目相关的不同角色而设计（如图4-2所示）。

- 架构师要为客户负责，满足他们的业务目标和约束条件；
- 架构师要为用户负责，使他们关心的功能需求和运行期质量属性得以满足；
- 架构师必须顾及处于协作分工“下游”的开发人员；
- 架构师还必须考虑“周边”的管理人员，为他们进行分工管理、协调控制和评估监控等工作提供清晰的基础。

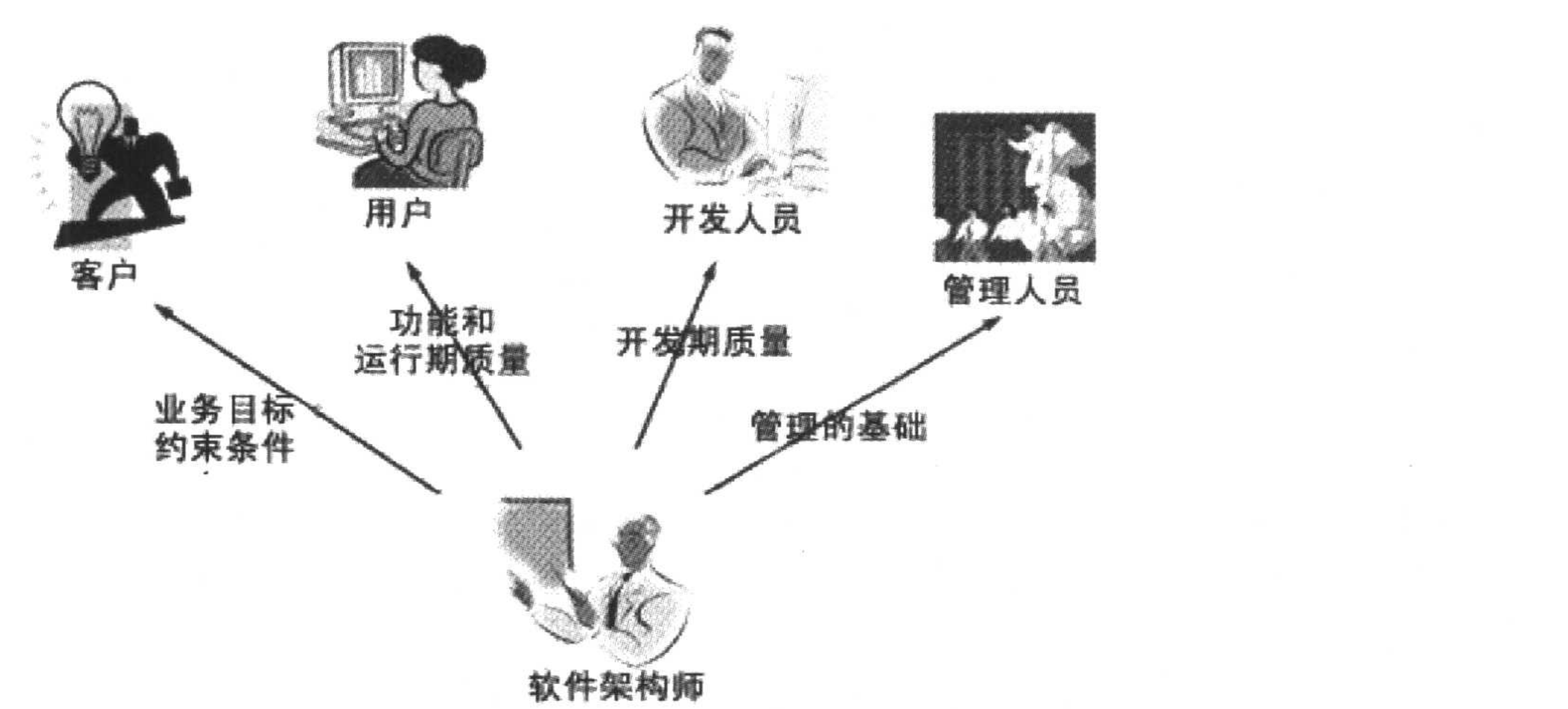


图 4-2 软件架构为谁而设计

一言以蔽之，软件架构师必须做到内外兼顾、各层并重（如图 4-3 所示）。只有这样，软件架构才能和它“包含了关于如何构建软件的一些最重要的设计决策”的“地位”相符。

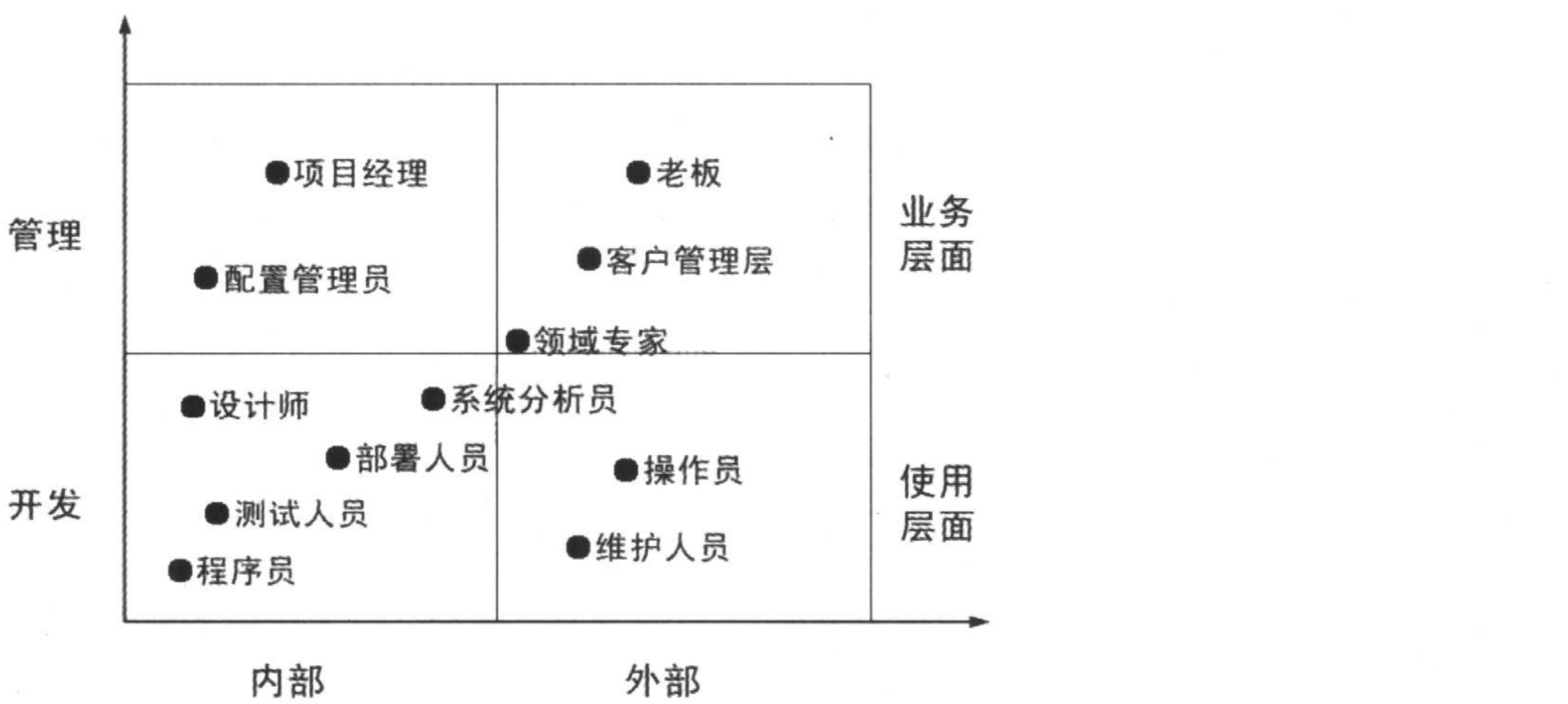


图 4-3 架构师应内外兼顾、各层并重

### 4.3 引入软件架构视图

下面着重讨论软件架构视图的概念及其在实践中的作用。



### 4.3.1 生活中的“视图”运用

先来看一个生活中运用视图的例子。

我们只有一个地球，但不同的时候我们会关心世界的不同问题：例如社会学家可能关心图 4-4 所示为世界人口分布图，而气候学家则更关心图 4-5 所示为世界年降水量分布图。

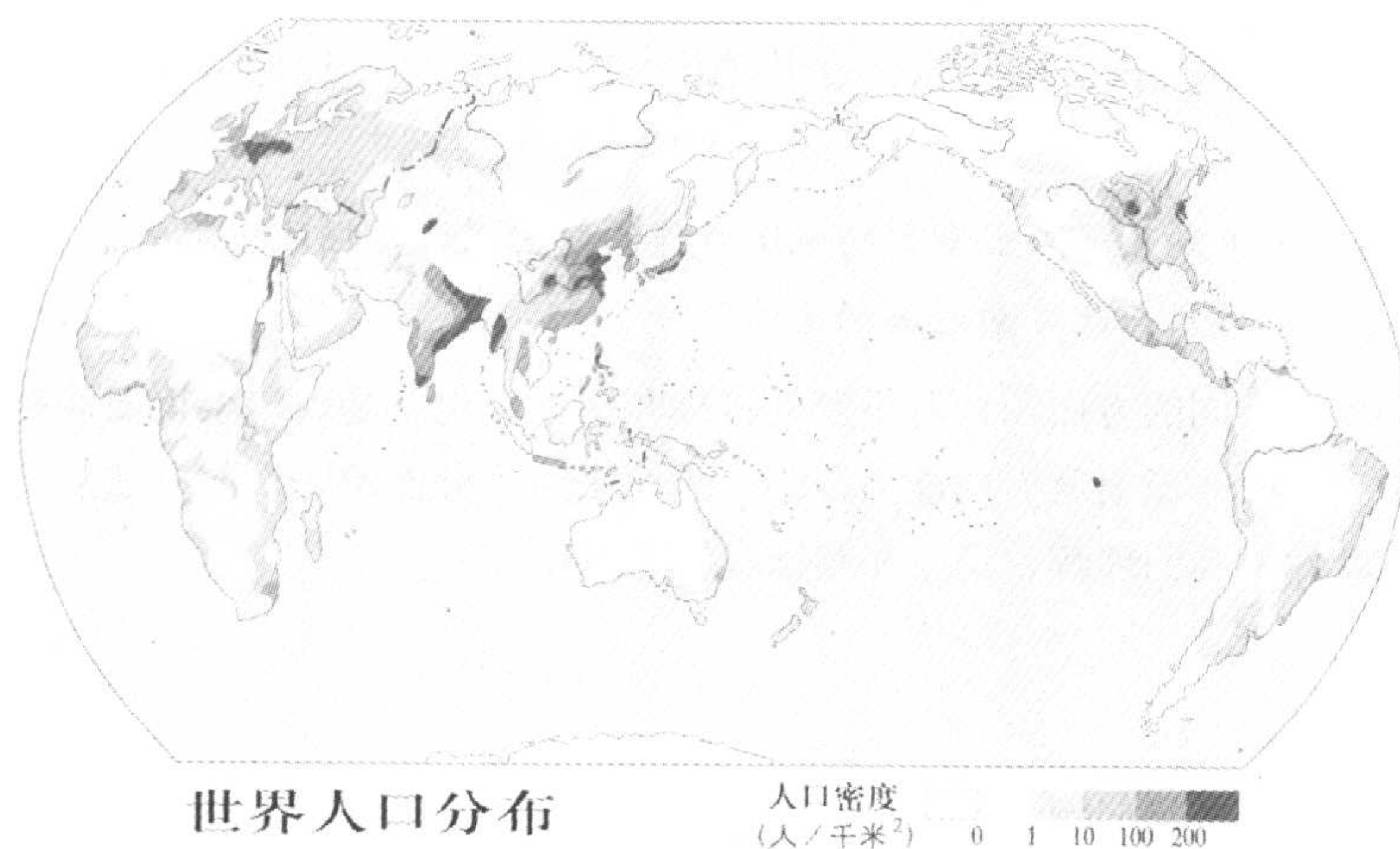


图 4-4 世界人口分布图（图片来源：www.dlpd.com）

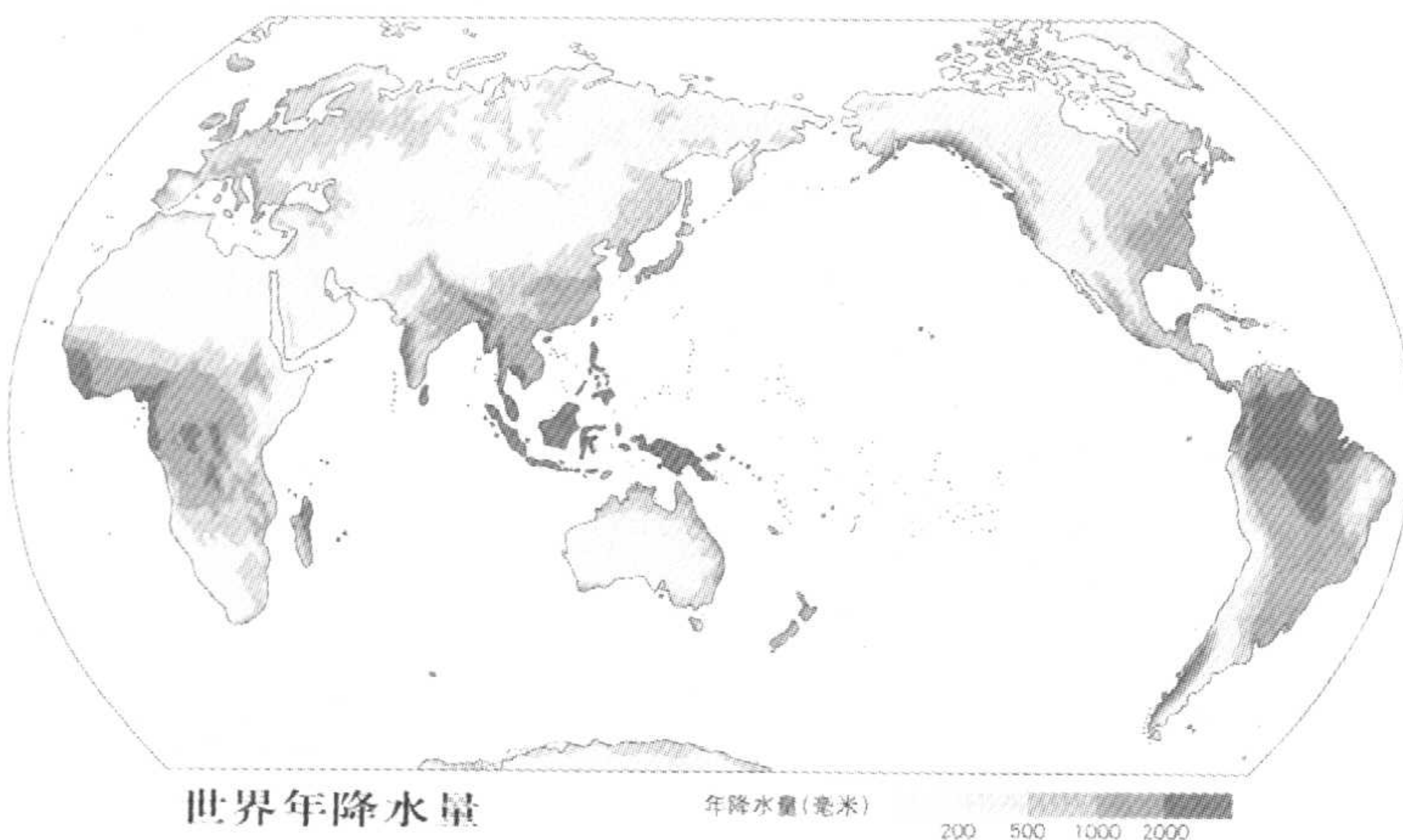


图 4-5 世界年降水量分布图（图片来源：www.dlpd.com）

其实上面就运用了“视图”作为手段，用不同的视图来刻画我们这个世界的不同方面。如果你喜欢，你完全可以将“世界人口分布图”称为“世界的人口分布视图”。这里引入视图的作用在于：世界地图的绘制者很难将不同的信息都绘制到同一幅图中；而看地图的人也希望有一幅地图是专门针对他的需要的。

当然，更进一步而言，同一事物的不同视图之间是有联系的。例如，从图 4-4 和 4-5 上看，除了南美洲之外基本都是降水量足的地方人口较密集。我们下面将讨论的软件架构视图也是这样，不同视图之间存在相互影响。

### 4.3.2 什么是软件架构视图

那么，什么是软件架构视图呢？Philippe Kruchten 在其著作《Rational 统一过程引论》中写道：

一个架构视图是对于从某一视角或某一点上看到的系统所作的简化描述，描述中涵盖了系统的某一特定方面，而省略了与此方面无关的实体。

软件架构的每个视图分别关注不同的方面，针对不同的目标和用途。也就是说，架构要涵盖的内容和决策太多了，超过了人脑“一蹴而就”的能力范围，因此采用“分而治之”的办法从不同视角分别设计；同时，也为软件架构的理解、交流和归档提供了方便。

### 4.3.3 多组涉众，多个视图

第一个问题，软件架构的表达与交流问题。

这是个很现实的问题。究其原因，由于角色和分工不同，整个软件团队以及客户等涉众各自需要掌握的技术或技能存在很大差异，为了完成各自的工作，他们需要了解整套软件架构决策的不同子集。所以，软件架构师应当提供不同的软件架构视图，以便交流和传递设计思想。例如，负责部署和运营维护的系统工程师最关心软件系统基于何种操作系统之上，依赖于哪些软件中间件，有没有群集或备份等部署要求，驻留在不同机器上的软件部分之间的通信协议是什么等决策；而开发人员则最关心软件架构方案中关于模块划分的决定，模块之间的接口如何定义，甚至架构指定的开发技术和现成框架是不是最流行的等问题；……；不一而足。

反过来，试想所有架构设计决策如果都混在一起表达会怎样？如此一来，不同的角色都会看到一个过于复杂的架构文档；更糟糕的是，谁都觉得难以理解这一软件架构，毕竟，将不同涉众关心的逻辑层（Layer）、物理层（Tier）、子系统、模块、接口、进程、线程、消息和协议等概念堆砌到一起，每个涉众都有可能看不懂了。

对此，Peter Herzum 等人在《Business Component Factory》一书中就曾指出：

总的来说，“架构”一词涵盖了软件架构的所有方面，这些方面紧紧地缠绕在一起，决定如何将之分割成部分和主题显得相当主观。既然如此，就必须引入“架构视点”作为讨论、归档和理解大型系统架构的手段（Generally speaking, the term architecture can be seen as covering all aspects of a software architecture. All its aspects are deeply intertwined, and it is really a subjective decision to split it up in parts and subjects. Having said that, the usefulness of introducing architectural viewpoints is essential as a way of



discussing, documenting, and mastering the architecture of large-scale systems ).

第二个问题，软件架构设计的问题。

这个问题更为关键。软件架构是个复杂的整体，架构师可以“一下子”把它想清楚吗？当然不能。有关思维的科学研究表明，越是复杂的问题越需要分而治之的思维方式。而每个软件架构视图关注软件架构不同的方面，使问题得以清晰化和简化，利于软件架构师完成架构设计工作。

软件架构设计中，会牵扯到很多概念和技术，例如逻辑层（Layer）、物理层（Tier）、子系统、模块、接口、进程、线程、消息和协议等。而利于软件架构视图的方法，可以一次只围绕少数概念和技术展开，分别着重研究软件架构的不同方面。

笔者注意到，大多数书籍中都强调多视图方法是软件架构归档的方法，却忽视了该方法对架构设计思维的指导作用。一方面，项目不同角色观察软件架构的视角各不相同，每个视角涉及的需求和技术也不尽相同，所以将软件架构视图用作架构归档的手段是顺理成章的；但更为重要的是，软件架构视图可以被相对独立地分析和设计，这就使软件架构师可以暂时撇开其他问题、专注于特定问题进行深入的分析设计。

也就是说，多重视图的软件架构不仅仅是架构归档的方式，更是架构设计的思维方法。

## 4.4 实践指南：逻辑架构与物理架构

在不同的架构设计方法中出现的软件架构视图种类很多，下面介绍最常用的两种架构视图：逻辑架构视图和物理架构视图。

当观察和描述事物大局的时候，逻辑架构和物理架构是最常用的角度。比如，以我们办公室里的局域网为例：从物理角度看，所有计算机“毫无区别”地连接到路由器上；而从逻辑角度看呢，就发现这些计算机是有区别的——一台计算机充当文件服务器，而其他计算机是可以访问服务器的客户机。如图 4-6 所示。

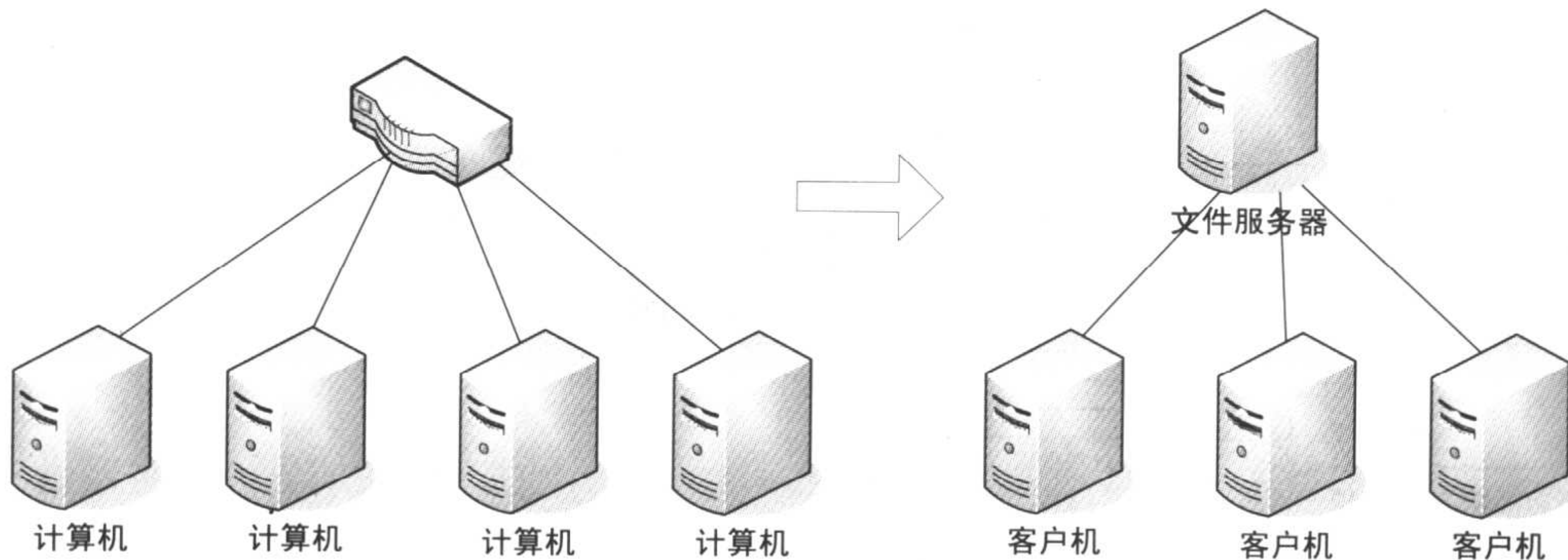


图 4-6 区分物理视角与逻辑视角



同样，在软件架构设计过程中，也可以通过区分软件的逻辑架构和物理架构，分别从不同的角度设计和描述软件架构。

#### 4.4.1 逻辑架构

软件的逻辑架构规定了软件系统由哪些逻辑元素组成以及这些逻辑元素之间的关系。

软件的逻辑元素一般指某种级别的功能模块，大到我们熟悉的逻辑层（Layer），以及子系统、模块，小到一个个的类。至于具体要分解到何种大小的功能模块才可结束软件架构设计，并不存在一个“一刀切”的标准——只要足够明确简单，能够分头开发就可以了。于是，在实践中我们往往将关键机制相关的架构设计部分明确到类，而一般功能则到模块甚至子系统的接口定义即可。

值得说明的是，功能模块有时容易识别，有时却比较隐含。而比较全面地识别功能块、规划功能块的接口和明确功能块之间的使用关系和使用机制，正是软件逻辑架构设计的核心任务所在。对此，Ivar Jacobson 曾有过极为形象的说法，“软件系统的架构涵盖了整个系统，尽管架构的有些部分可能只有‘一寸深’”。

图 4-7 展示了一个网络设备管理系统逻辑架构设计的一部分，我们借此来举例说明软件逻辑架构设计的三大核心任务：

- 识别功能块
- 规划功能块的接口
- 明确功能块之间的使用关系和使用机制

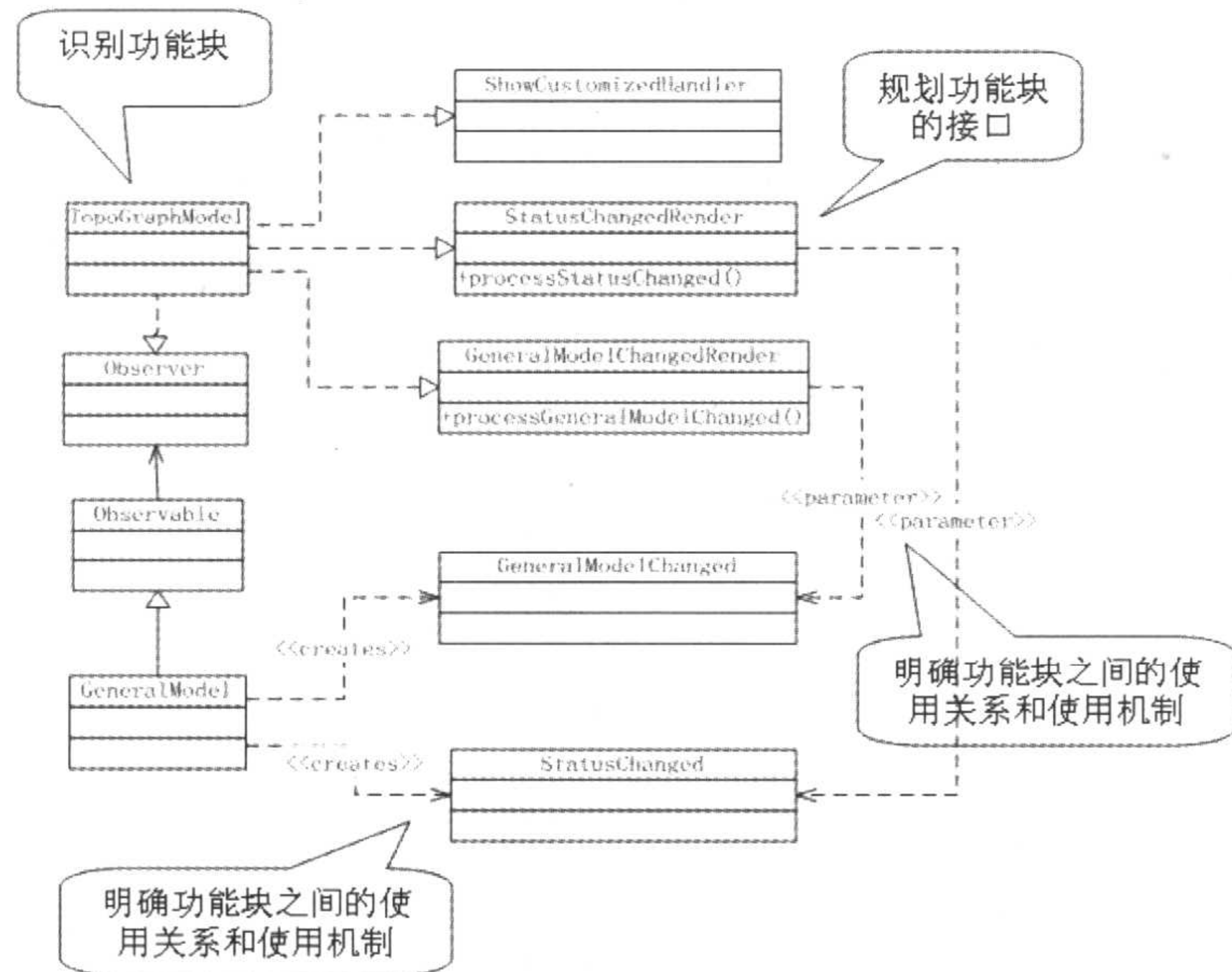


图 4-7 软件逻辑架构设计的核心任务

软件的逻辑架构是架构设计思维的重要方法。今天，用例技术已经成为捕获功能需求的事实标准，所以逻辑架构的设计往往是从用例分析开始的。基于用例的分析方法使逻辑架构的设计变得比较有序——通过对每个关键用例的分析，从逻辑上将用例实现为一组功能块的特定组合，最后综合这些用例分析成果，得到整个软件系统的逻辑架构。而在用例分析方法产生之前，功能模块的确定多多少少带有些“硬”想出来的味道，特别是并不直接承载业务功能的模块有时比较容易遗漏，直到大规模编程实现阶段才发现。

## 4.4.2 物理架构

软件的物理架构规定了组成软件系统的物理元素，这些物理元素之间的关系，以及它们部署到硬件上的策略。

物理架构可以反映出软件系统动态运行时的组织情况。此时，上述物理架构定义中所提及的“物理元素”就是进程、线程以及作为类的运行时实例的对象等，而进程调度、线程同步、进程或线程通信等则进一步反映物理架构的动态行为。

随着分布式系统的流行，“物理层（Tier）”的概念大家早已耳熟能详。物理层和分布有关，通过将一个整体的软件系统划分为不同的物理层，可以把它部署到分布在不同位置的多台计算机上，从而为远程访问和负载均衡等问题提供了手段。当然，物理层是大粒度的物理单元，它最终是由粒度更小的组件、模块和进程等单元组成的。

物理架构的应用很广泛。例如，架构设计中可能需要专门说明数据是如何产生、存储、共享和复制的，这时可以利于物理架构，展示软件系统在运行期间数据是由哪些运行时单元产生以及如何产生的，数据又如何被使用，如何被存储，哪些数据需要跨网络复制和共享等方面的设计决策。

由于人们对组成软件系统的“物理元素”存在不同看法（如图 4-8 所示），所以在实践中物

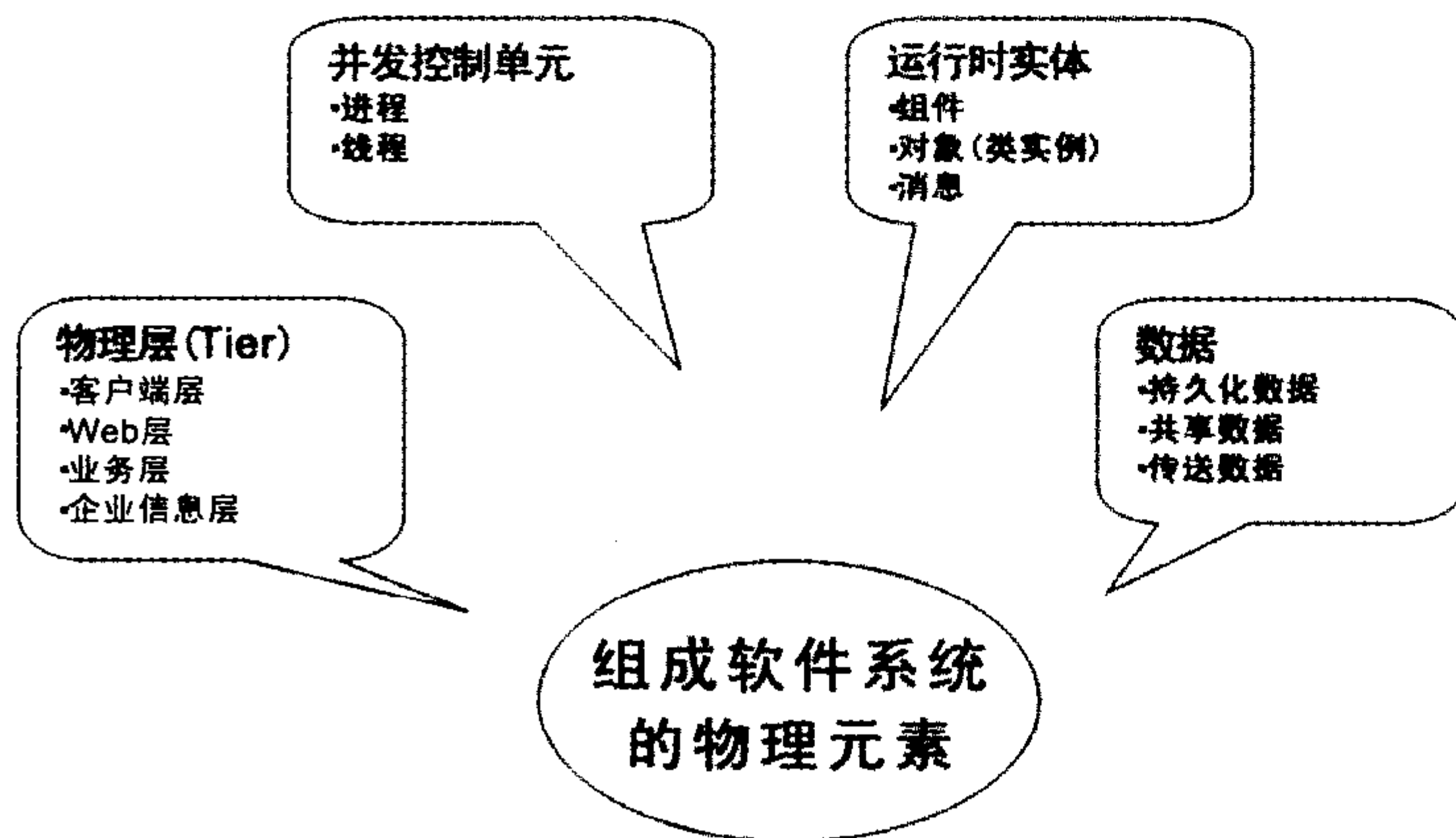


图 4-8 对“物理元素”的不同看法

理架构的用法比较宽泛，不同的人所认为的物理架构也可能不尽相同。因此，我们在交流和实践的过程中，应注意区分物理架构所指为何。（也正是因为这个原因，实践中所采用的基于多视图的架构设计方法往往包含更多的视图，从而使每个架构视图的职责更加明确。例如，本书第 5 章将要讲述的“架构设计的 5 视图法”就包含了逻辑架构、开发架构、运行架构、数据架构和物理架构。）

4.4.3 从逻辑架构和物理架构到设计实现

逻辑架构和物理架构是软件架构设计的重要方面。逻辑架构致力于将软件系统分解成不同的逻辑单元，并规定这些逻辑单元之间的交互接口和交互机制。物理架构则更重视软件系统运行时的动态结构，以及组成软件系统的目标程序如何部署到硬件上。

在后续的详细设计和编程实现中，将贯彻和利用逻辑架构和物理架构设计中制定的架构决策，如图 4-9 所示。

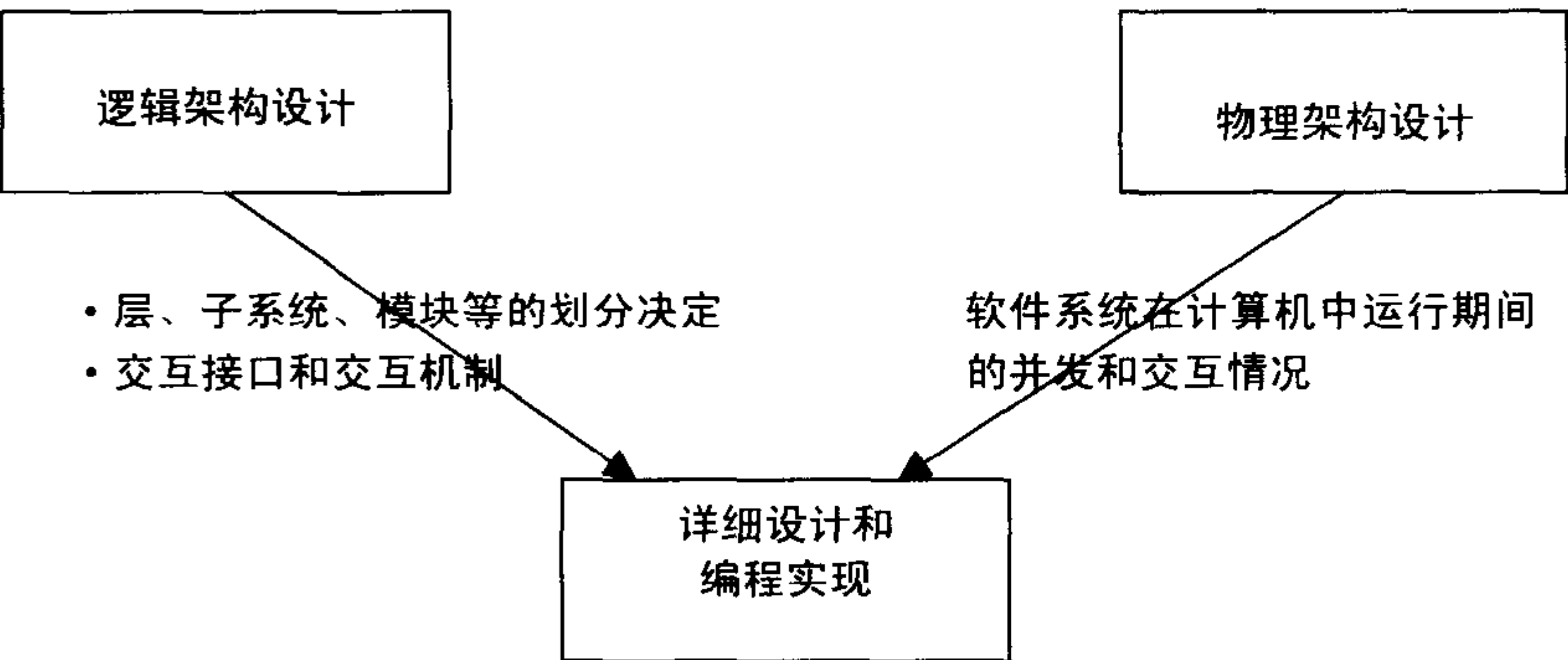


图 4-9 逻辑架构和物理架构对后续开发的作用

逻辑架构中关于职责划分的决策，体现为层、子系统和模块等的划分决定，从静态视角为详细设计和编程实现提供切实的指导；有了分解就必然产生协作，逻辑架构还规定了不同逻辑单元之间的交互接口和交互机制，而编程工作必须实现这些接口和机制。

所谓交互机制，是指不同软件单元之间交互的手段。交互机制的例子有：方法调用、基于 RMI 的远程方法调用、发送消息等。

至于物理架构，它关注的是软件系统在计算机中运行期间的情况。物理架构设计方案中规定了软件系统如何使用进程和线程完成期望的并发处理，进程线程这些主动对象（Active Object）会调用哪些被动对象（Passive Object）参与处理，交互机制（如消息）为何等问题，从而为详细设计和编程实现提供了工作目标的动态视图。



## 4.5 设备调试系统案例：领会逻辑架构和物理架构

下面通过一个实际案例的分析，来帮助领会逻辑架构和物理架构这两种架构视图对架构设计的指导作用。

### 4.5.1 设备调试系统案例简介

该案例是某型号设备调试系统。设备调试员通过使用该系统，可以察看设备状态（设备的状态信息由专用的数据采集器实时采集），发送调试命令。该系统的用例图如图 4-10 所示。

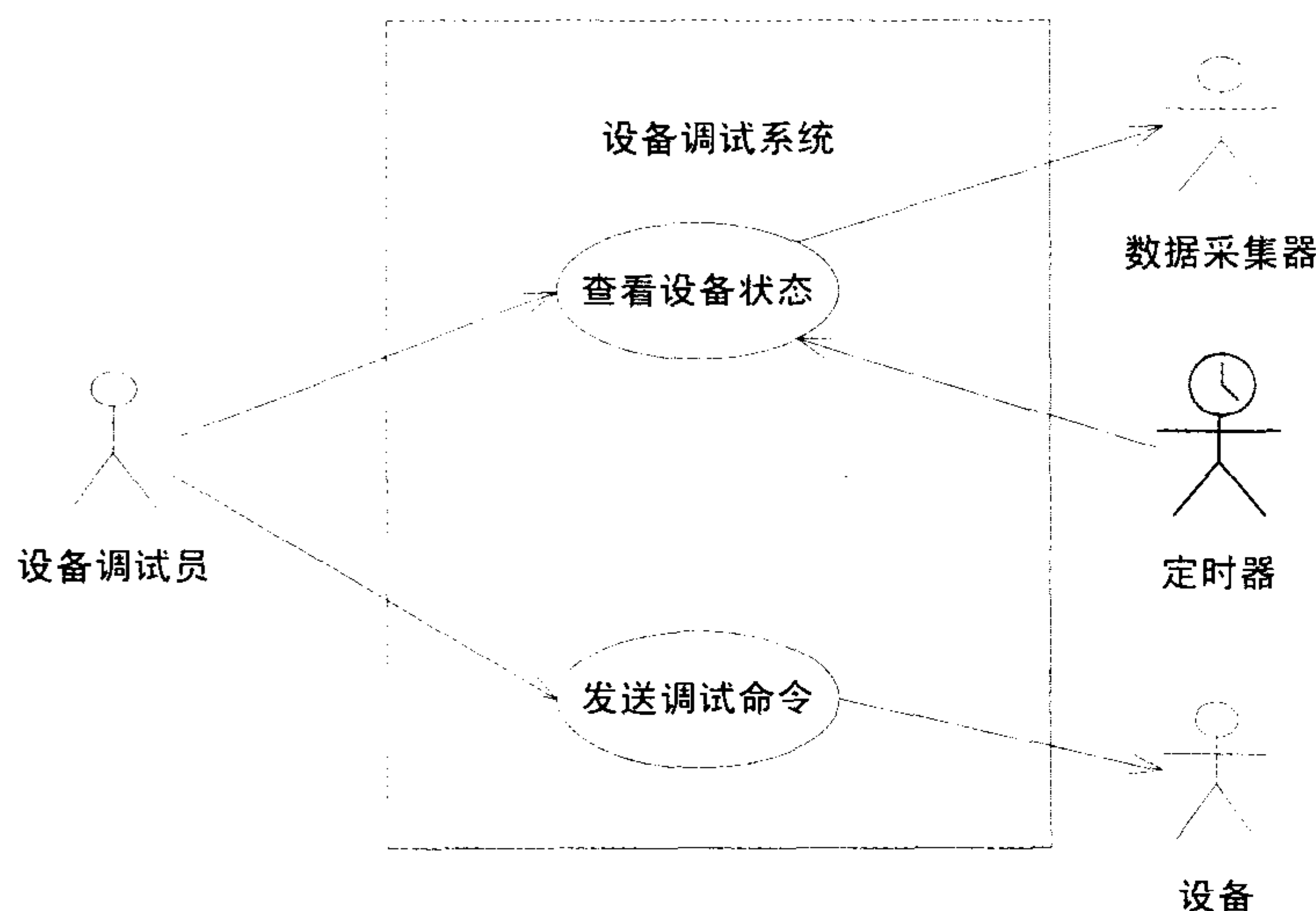


图 4-10 设备调试系统的用例图

### 4.5.2 逻辑架构设计

首先根据功能需求进行初步设计，进行大粒度的职责划分。如图 4-11 所示。

之后，还有很多与逻辑架构设计相关的工作要做。例如，图 4-12 所示的 CRC 卡描述了上面的三层架构每一层的职责与协作者：

- 应用层负责设备状态的显示，并提供模拟控制台供用户发送调试命令；
- 应用层使用通讯层和设备控制层进行交互，但应用层不知道通讯层的细节；
- 通讯层负责在 RS232 协议之上实现一套专用的“应用协议”；
- 当应用层发送来包含调试指令的协议包，由通讯层负责按 RS232 协议将之传递给设备控制层；
- 当设备控制层发送来原始数据时，由通讯层将之解释成应用协议包发送给应用层；

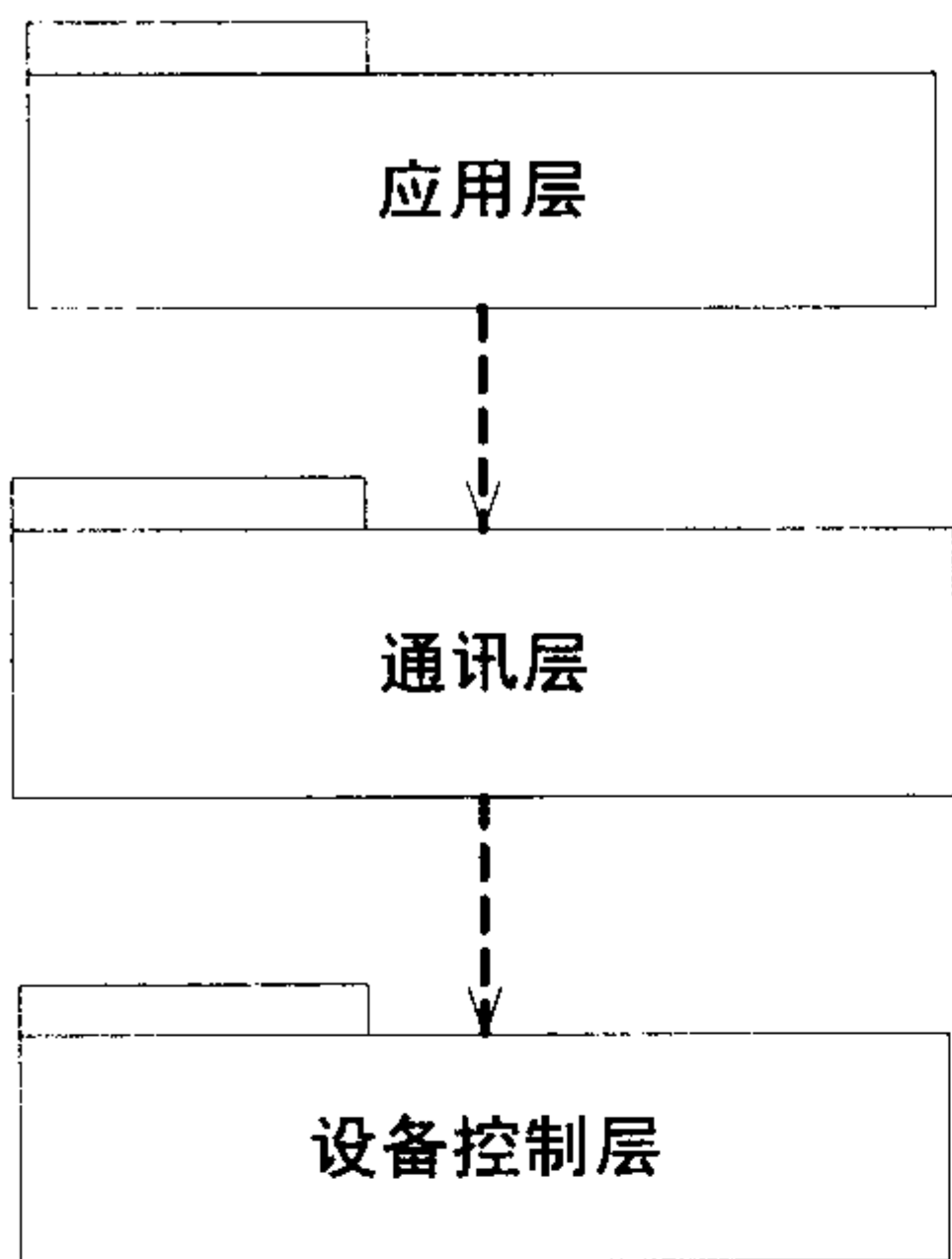


图 4-11 设备调试系统的逻辑架构

- 设备控制层负责对调试设备的具体控制，以及高频度地从数据采集器读取设备状态数据；
- 设备控制指令的物理规格被封装在设备控制层内部，读取数采器的具体细节也被封装在设备控制层内部。

<div><div>应用层</div><div>职责<ul style="list-style-type: none"><li>●负责设备状态的显示</li><li>●提供模拟控制台供用户发送调试命令</li><li>●使用通讯层和设备控制层进行交互</li></ul></div></div> <div>协作者<ul style="list-style-type: none"><li>●通讯层</li></ul></div>	
<div><div>通讯层</div><div>职责<ul style="list-style-type: none"><li>●负责在RS232协议之上实现一套专用的“应用协议”</li><li>●当应用层发送来包含调试指令的协议包，负责按RS232协议将之传递给设备控制层</li><li>●当设备控制层发送来原始数据，将之解释成应用协议包发送给应用层</li></ul></div></div> <div>协作者<ul style="list-style-type: none"><li>●设备控制层</li></ul></div>	
<div><div>设备控制层</div><div>职责<ul style="list-style-type: none"><li>●负责对调试设备的具体控制</li><li>●高频度地从数据采集器读取设备状态数据</li><li>●将指令按设备控制指令的物理规格发送给设备</li></ul></div></div> <div>协作者</div>	

图 4-12 用 CRC 卡描述每层的职责和协作者

### 4.5.3 物理架构设计

软件最终要驻留、安装或部署到硬件才能运行。软件的物理架构关注“目标程序及其依赖的运行库和系统软件”最终如何安装或部署到物理机器，以及如何部署机器和网络来配合软件系统的可靠性、可伸缩性等要求。

多个逻辑层（Layer）可以映射到一个物理层（Tier），这是很多从事分布式开发的读者都了解的。在进行设备调试系统的物理架构设计时，也体现了这一点。如图 4-13 所示，设备调试系统共包含 2 个物理层：桌面部分和嵌入部分。作为逻辑层的应用层和通讯层最终将成为桌面部分，而设备控制层最终成为嵌入部分。

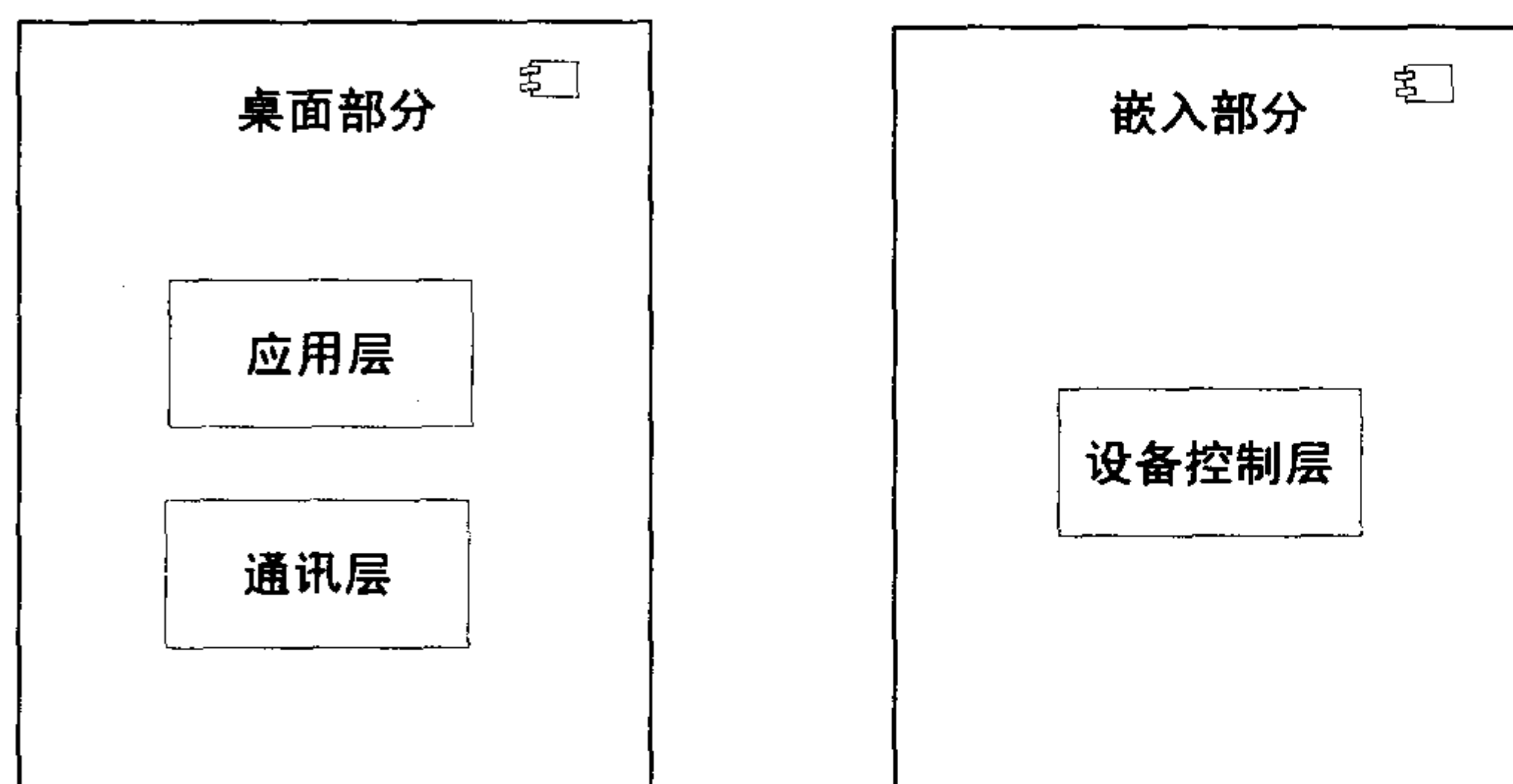


图 4-13 逻辑层（Layer）到物理层（Tier）的映射

物理层作为组成软件系统的物理单元，最终又要映射到具体的硬件，这也是物理架构设计要考虑的，对于分布式软件系统的设计而言尤其不可或缺。图 4-14 展示了这一点。可以看出，设备控制部分驻留在调试机中（调试机是专用单板机），而桌面部分以常见的“Windows 可执行程序”的形式运用于 PC 机之上。

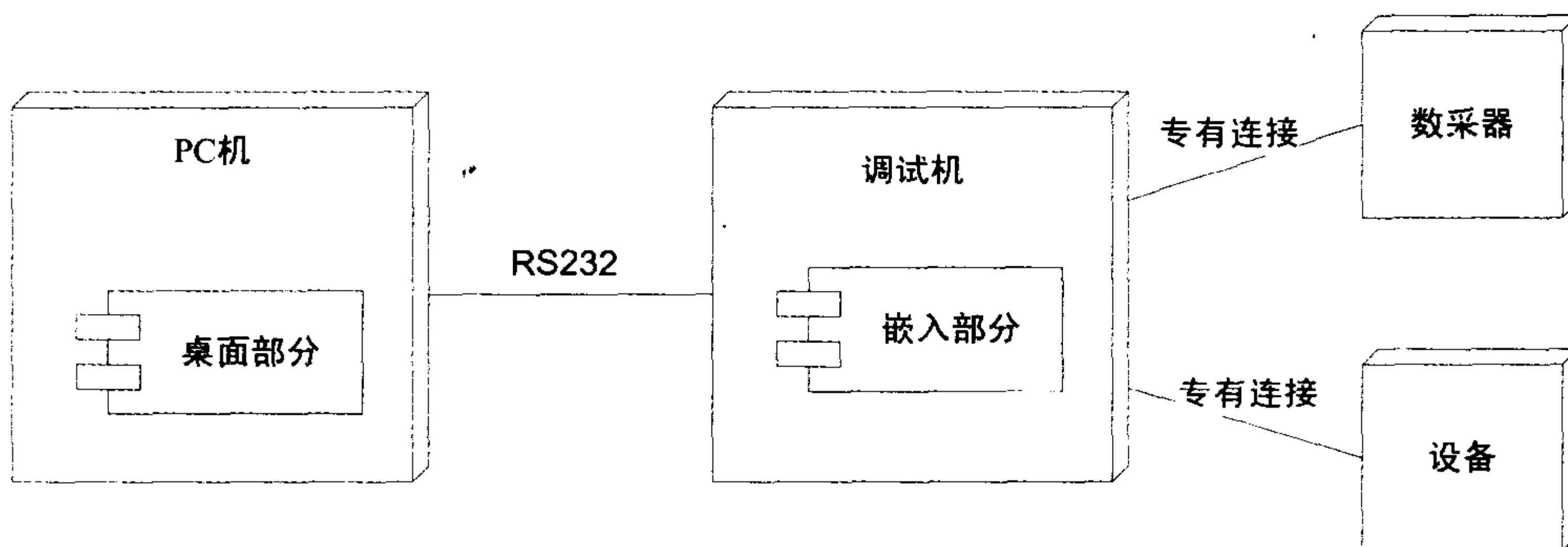


图 4-14 设备调试系统架构的物理架构

我们还可能根据具体情况的需要，通过物理架构视图更明确地表达具体目标模块及其通讯结构，如图 4-15 所示。



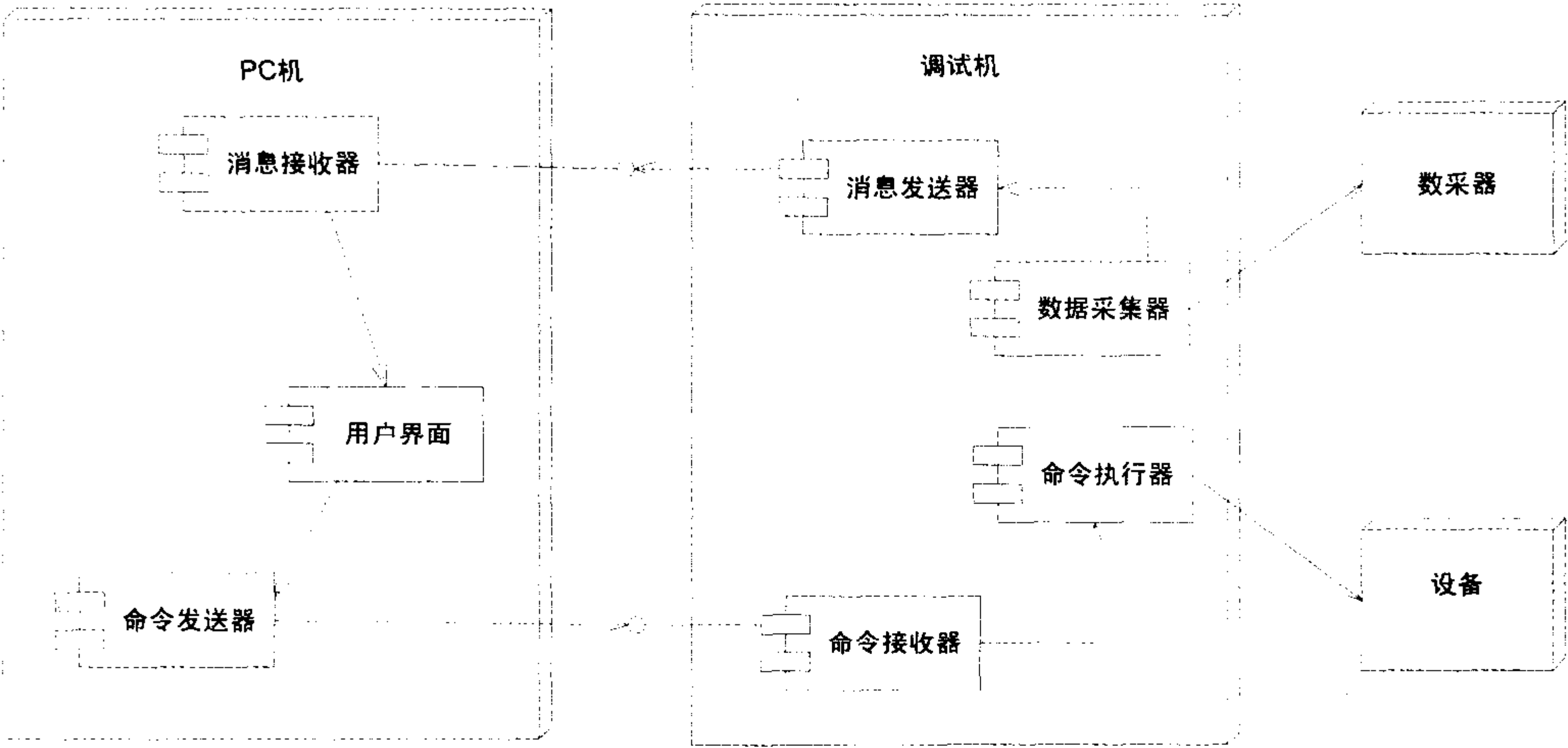


图 4-15 设备调试系统的物理架构

## 4.6 总结与强调

从简单系统到复杂系统的变化，对架构设计的冲击决不仅仅是量变的问题。本章引入了软件架构视图的概念，有助于软件架构师控制架构设计的复杂性。

软件架构视图的概念和软件架构基本概念是完全相容的，后者针对软件系统的整体目标，而前者针对特定目标子集，这样一来，重点突出了，问题明确了，软件架构师的经验也就活跃了起来。

逻辑架构和物理架构相分离的设计方法在软件实践中比较常用。逻辑架构和物理架构是软件架构设计的重要方面。逻辑架构致力于将软件系统分解成不同的逻辑单元，并规定这些逻辑单元之间的交互接口和交互机制。物理架构则更重视软件系统运行时的动态结构，以及组成软件系统的目标程序如何部署到硬件上。

## 第 5 章 架构设计的 5 视图法

---

“笑嘻嘻的小猫咪，”爱丽丝问道，“请你告诉我该走哪条路？”“那取决于你想去何方。”小猫回答说。

——路易斯·卡罗尔，《爱丽丝漫游仙境》

不同的视图支持不同的目标和用途。

——Paul Clements，《软件架构编档》

如果选择视图的工作没有做好，或者以牺牲其他视图为代价只注重一个视图，就会冒掩盖问题以及延误解决问题（这里的问题是指那些最终会导致失败的问题）的风险。

——Grady Booch，《UML 用户指南》

方法如路标。

如果地形复杂，我们会迷路，但如果有了路标，将利于我们找到前进的方向。好的方法也是如此，它对实践者有启发和指引作用。

软件架构师有许多工作要做：

- 要满足性能、持续可用性等方面的需求，架构师必须深入研究软件系统运行期间的情况、制定相应的设计决策，这些需求被称为软件的“运行期质量属性”；
- 而要满足可扩展性、可重用性等方面的需求，则要求架构师深入研究软件系统开发期间的情况，制定相应的设计决策，这些需求被称为软件的“开发期质量属性”；
- 约束是一类特殊的需求，带有一定强制性，架构师制定的架构决策必须满足这些限制；
- 为了满足功能需求，架构师必须规划组成软件系统的所有模块，为他们分配不同职责，使这些模块可以通过协作完成功能需求。

基于多视图的架构设计方法恰好可以帮助软件架构师完成上述工作。这就是本章的主题。

## 5.1 架构设计的 5 视图法

之所以要采用架构设计的 5 视图法，这和软件需求的复杂性有关。软件架构师必须明确区分功能需求、约束、运行期质量属性和开发期质量属性等不同种类的需求，这些需求对架构设计的影响是截然不同。基于多视图的架构设计方法在一定程度上将各类需求分别对待，通过不同的架构设计视图分别满足它们，从而确保重要的需求一一被满足。

图 5-1 展示了架构设计 5 视图方法的概况，它包含了逻辑架构、开发架构、运行架构、物理架构、数据架构等 5 个架构设计视图。

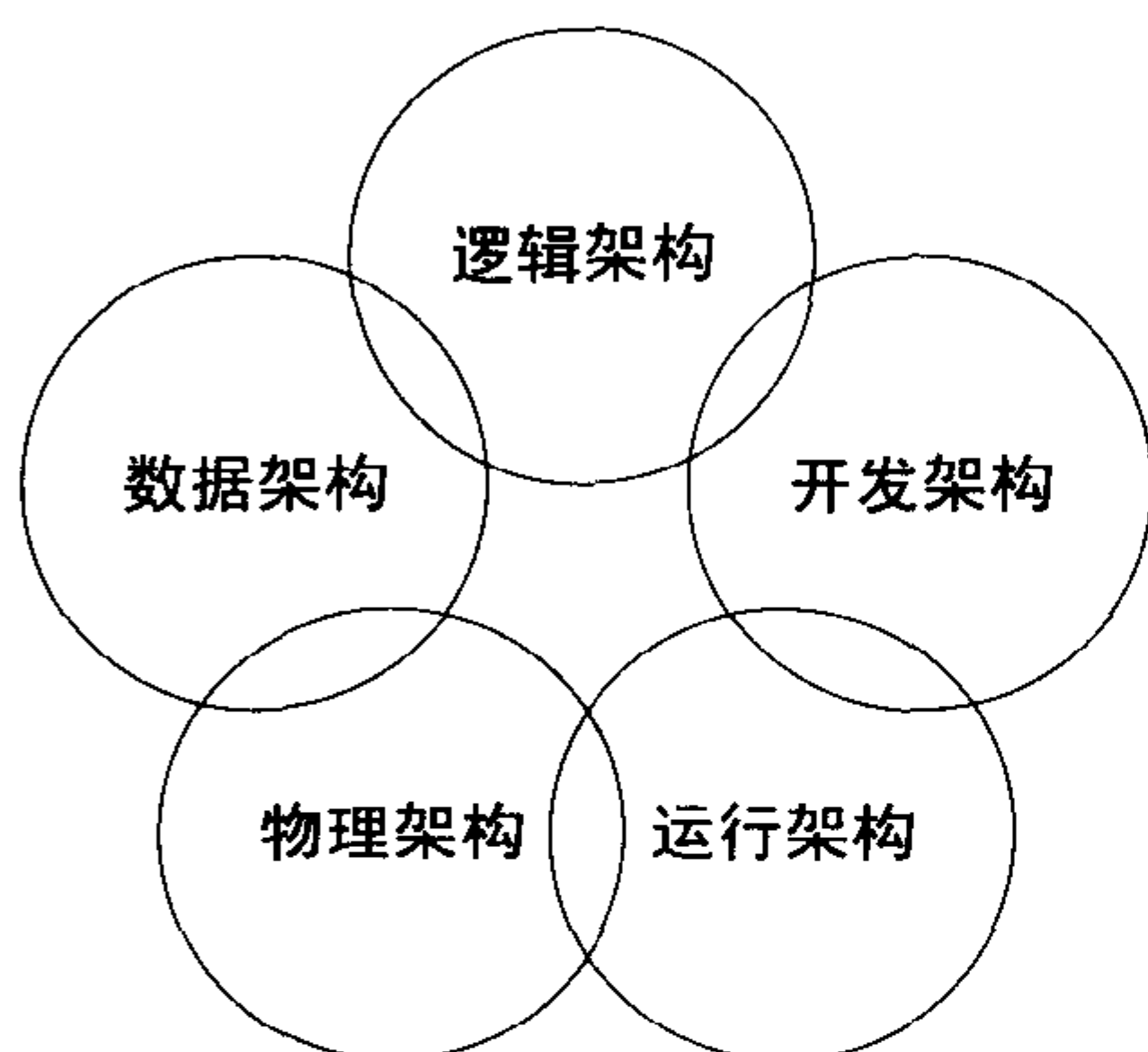


图 5-1 架构设计的 5 视图方法

**逻辑架构。**逻辑架构关注功能，不仅包括用户可见的功能，还包括为实现用户功能而必须提供的“辅助功能模块”；它们可能是逻辑层、功能模块和类等。

**开发架构。**开发架构关注程序包，不仅包括要编写的源程序，还包括可以直接使用的第三方 SDK 和现成框架、类库，以及开发的系统将运行于其上的系统软件或中间件。开发架构和逻辑架构之间可能存在一定的映射关系：比如逻辑架构中的逻辑层一般会映射到开发架构中的多个程序包；再比如开发架构中的源码文件可以包含逻辑架构中的一到多个类（在 C++ 里一个源码文件可以包含多个类，即使在 Java 里一个源码文件也可以同时包含一个类和几个内部类）。

**运行架构。**运行架构关注进程、线程、对象等运行时概念，以及相关的并发、同步、通信等问题。运行架构和开发架构的关系：开发架构一般偏重程序包在编译时期的静态依赖关系，而这些程序运行起来之后会表现为对象、线程、进程，运行架构比较关注的是这些运行时单元的交互问题。

**物理架构。**物理架构关注“目标程序及其依赖的运行库和系统软件”最终如何安装或部署到物理机器，以及如何部署机器和网络来配合软件系统的可靠性、可伸缩性等要求。物理架构和运



行架构的关系：运行架构特别关注目标程序的动态执行情况，而物理架构重视目标程序的静态位置问题；物理架构还要考虑软件系统和包括硬件在内的整个 IT 系统之间是如何相互影响的。

数据架构。数据架构关注持久化数据的存储方案，不仅包括实体及实体关系的数据存储格式，还可能包括数据传递、数据复制和数据同步等策略。数据架构和物理架构的关系：对于很多集成系统，数据需要在不同系统之间传递、复制和暂存，这往往要涉及到不同的物理机器；也就是说，如果需要，可以把数据放在物理架构之中考虑，以便体现集成系统的数据分布与传递特征。

5 视图架构设计方法的不同视图，所重点针对的需求类型不同。如图 5-2 所示。

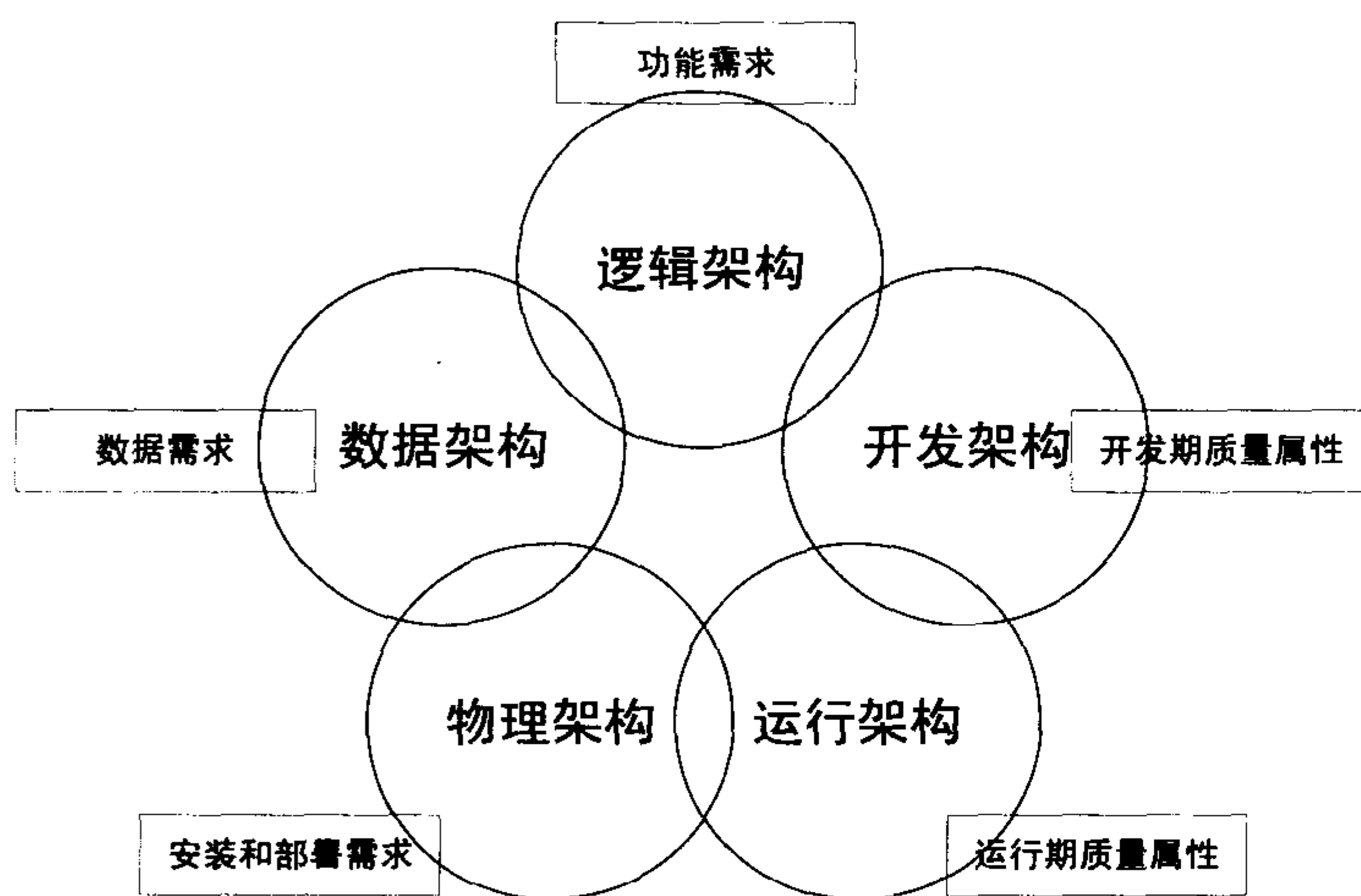


图 5-2 不同视图所重点针对的需求类型不同

逻辑架构的设计着重考虑功能需求——系统应当向用户提供什么样的服务。逻辑架构的关注点主要是行为或职责的划分。如果使用 UML 来描述架构的逻辑架构，则该视图的静态方面由包图、类图、对象图来描述，动态方面由序列图、协作图、状态图和活动图来描述。

开发架构的设计着重考虑开发期质量属性，例如可扩展性、可重用性、可移植性、易理解性和易测试性等。开发架构的关注点是在软件开发环境中软件模块的实际组织方式，具体涉及源程序文件、配置文件、源程序包、编译（或许还需要打包）后的目标文件和第三方库文件等。如果使用 UML 来描述架构的逻辑架构，则该视图可能包括包图、类图和组件图等。

运行架构的设计着重考虑运行期质量属性，例如性能、可伸缩性、持续可用性和安全性等。运行架构的关注点是系统的并发与同步等问题，这势必涉及到进程和线程等技术。如果使用 UML 来描述架构的运行架构，则该视图的静态方面由包图、类图（其中主动类非常重要）和对象图（其中主动对象非常重要）等来说明关键运行时概念的结构关系。动态方面由序列图、协作

图等来说明关键交互机制。

物理架构的设计着重考虑“安装和部署需求”。物理视图描述运行软件的计算机、网络 and 硬件设施等情况，还包括如何将软件包部署（如果是嵌入式系统则是烧写）到这些硬件资源上，以及它们运行时的配置情况。物理架构的关注点是软件的目标单元如何映射到硬件；另外，由于一部分运行时质量属性需要硬件或网络的支持，所以物理架构还应关注相关的可靠性、可伸缩性、持续可用性、性能和安全性等方面。如果使用 UML 来描述架构的运行架构，则该视图可能包括部署图和组件图。

数据架构的设计着重考虑“数据需求”。数据架构的关注点是持久化数据的组织、数据传递、数据复制和数据同步等策略。数据架构的描述一般用 E-R 图和数据流图表示。当采用 UML 时，可以用特定版型（Stereotype）的 UML 类图替代 E-R 图，采用带对象流的活动图替代数据流图。

有些人倾向于认为逻辑架构视图反映架构的静态方面，运行架构视图反映架构的动态方面，这是不全面的。逻辑架构也好，运行架构也好，它们都有静态方面和动态方面；逻辑架构的静态方面关注抽象职责的划分，其动态方面关注承担不同职责的逻辑单元之间的交互；运行架构的静态方面关注软件的运行时单元的结构，其动态方面则关注运行时单元之间的交互机制。

## 5.2 实践中的5视图方法

在运用5视图方法进行架构设计时需要注意两方面的问题，以适应实际情况的需要。

一个是多个架构视图之间的同步问题。

不同软件架构视图之间是独立的吗？不完全是。因为它们分别反映同一个软件系统的不同设计方面，它们最终合在一起才是完整的架构设计方案，所以不同架构视图之间势必有相互支撑的关系。所谓保持架构视图之间的同步，就是要保证不同视图之间是相互解释而不是相互矛盾的。

例如，逻辑架构中的一个逻辑层到了开发视图中可能变成了几个具体的程序包，而程序包编译（可能还包括打包）后的目标程序的部署（对嵌入式系统可能是烧写）是物理架构所要考虑的。再如物理架构中可能会涉及数据的分布和传递备份，这就需要数据架构中有相应数据的定义和结构信息等。

另一个是架构视图的数量问题。

正如上面所讨论的，视图之间的同步是多视图方法的“开销”所在，因此一般而言，我们应该限制软件架构视图的数量。我们常常遇到的情况包括：

- 有些软件系统并不涉及持久化数据，那么就不需要进行数据架构设计；
- 运行于个人电脑之上的孤立的桌面应用，由于不涉及程序的分布问题，所以往往不需



要单独的物理架构设计；

- 对于业务逻辑较简单的软件（它们的计算逻辑未必简单），在实践中常常将逻辑视图和开发视图合二为一，此时“逻辑层”的概念可以和“程序包”的概念等同。

当然，如果需要，可以引入新的架构视图，从而更加突出和明确地制定和表达特定方面的架构决策。就拿安全性来说吧，如果安全性对软件系统来说极为关键，就可以引入单独的安全架构视图。在很大程度上，安全架构视图是其他视图中的安全性相关内容的汇集，例如：会话管理和授权管理等逻辑单元的引入来自逻辑视图；采用何种第三方加密算法包来自开发视图；消息的验证和转发涉及到运行视图；SSL 等安全通信协议的使用策略来自物理视图；对数据库采用的专有安全限制策略来自数据视图。

### 5.3 办公室里的争论：回顾与落实

在第 4 章中，本书描述了一个发生在办公室里的争论，说明了不同涉众往往从不同的视角看待软件架构。而架构设计的 5 视图方法可以很好地解决他们的问题，下面通过表格的形式给予说明（如表 5-1 所示）。

表 5-1 涉众关心的问题与相应架构视图

涉众及视角	相应的架构视图
程序员说，软件架构就是要决定需要编写哪些类、使用哪些现成框架（Framework）	开发架构
程序经理说，软件架构就是模块的划分和接口的定义	逻辑架构
系统分析员说，软件架构就是为业务领域对象的关系建模	逻辑架构
配置管理员说，软件架构就是开发出来以及编译过后的软件到底是个啥结构	开发架构
数据库工程师说，软件架构规定了持久化数据的结构，其他一切都不过是对数据的操作而已	数据架构
部署工程师说，软件架构规定了软件部署到硬件的策略	物理架构
用户说，软件架构就是决定一个个功能子系统如何划分	逻辑架构

### 5.4 案例：再谈设备调试系统

在第 4 章中，我们从逻辑架构和物理架构两个方面对设备调试系统进行了架构设计。本章将根据架构设计 5 视图法的指导，继续把设备调试系统的架构设计活动深入下去。



5.4.1 根据需求决定引入哪些架构视图

经过研制方和委托方的紧密配合，最终确定的需求（经简化）可以总括地用表 5-2 来表示。

表 5-2 设备调试系统的需求

非功能需求			功能需求
约束	运行期质量属性	开发期质量属性	
程序的嵌入式部分必须用 C 语言开发 一部分开发人员没有嵌入式开发经验	高性能	易测试性	察看设备状态 发送调试命令

作为软件架构师，将这份需求与上一章给出的设计对照之后可能会有如下思想活动：

- 约束是必须遵守和考虑到的，每个架构设计视图都应注意这一点；
- 高性能作为运行期质量属性需求，应多多研究设备调试系统的运行时结构与交互，制定出高性能所需的设计决策；
- 易测试性的确很有必要，毕竟牵扯到硬件设备的问题，如果这次搞硬件的同事再迟迟交不出合格的设备，软件这边的进度可不能跟着拖延，所以要把与硬件相关和通信相关的模块都独立出来。

于是，软件架构师决定再引入开发架构和运行架构的设计，而数据架构设计是不需要的，因为没有持久化数据。

5.4.2 开发架构设计

软件系统的开发架构视图应当为开发人员提供切实的指导。如果开发架构设计不足，就会造成一些影响全局的设计决策被“漏”到后边，等到大规模开发时，由程序员碰头儿临时决定，软件质量必然下降甚至导致项目失败。

其中，采用哪些现成框架，哪些第三方 SDK 乃至哪些中间件平台，都应该考虑是否由软件架构的开发视图确定下来。图 5-3 展示了设备调试系统（一部分）的开发架构：应用层将基于 MFC 设计实现，而通讯层采用了某串口通讯的第三方 SDK。

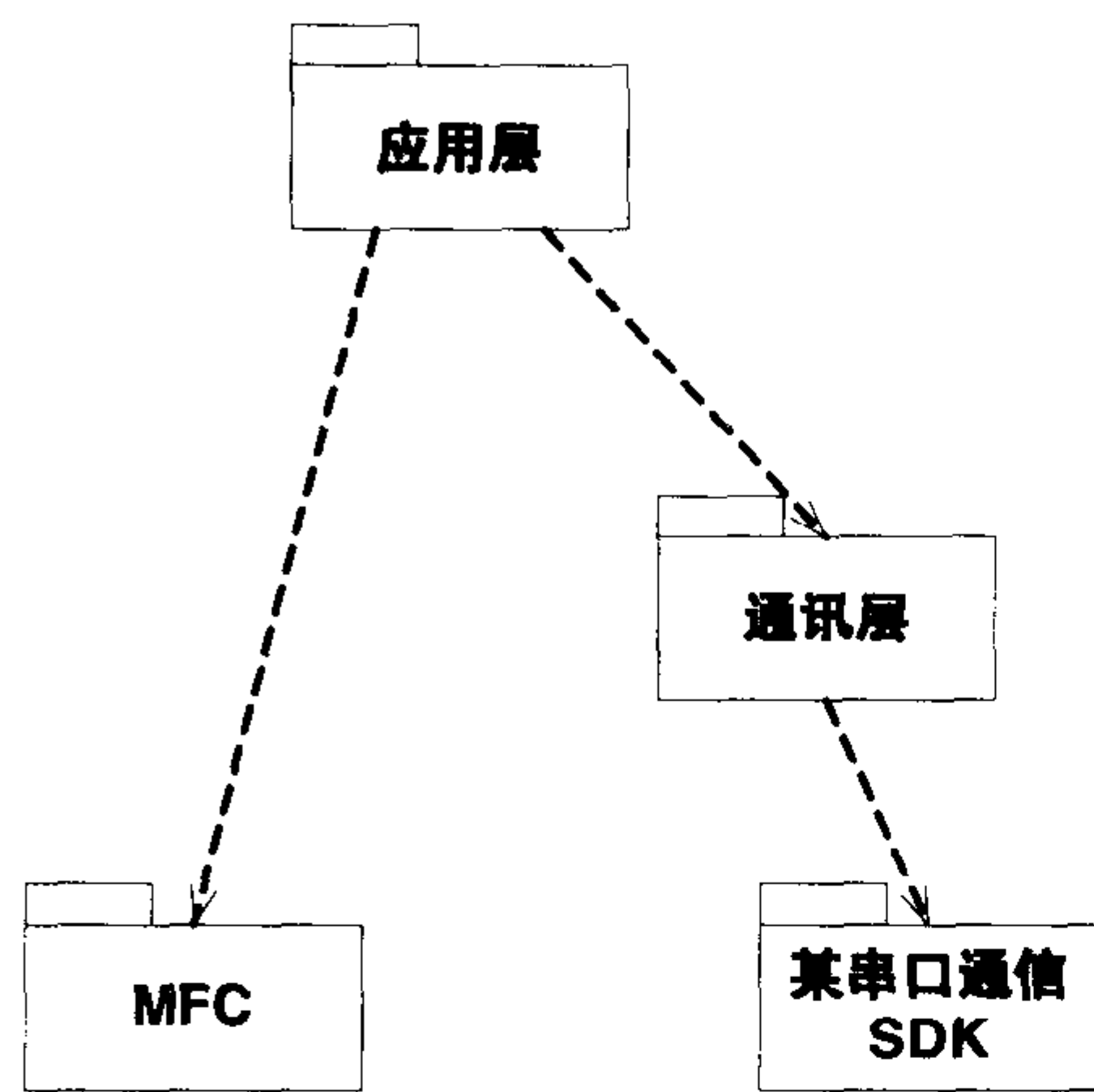


图 5-3 设备调试系统的开发架构

同时，每个架构设计视图都应当注意满足约束性需求的要求。既然“一部分开发人员没有嵌入式开发经验”，那么架构设计方案应弥补这一点所造成的影响，让更多开发人员都清楚我们的架构规划。图 5-4 展示了整个系统是如何编译的：桌面部分的应用程序作为 VC++项目，最终的cpp 文件会被编译成一个名为 pc-module.exe 的标准 Windows 应用程序；而嵌入式部分是一个 C51 项目，c 文件和 asm 文件编译后的结果是可供烧写的 rom-module.hex 文件。这个全局性的描述无疑对没有经验的开发人员提供了实感，利于更全面地理解系统的软件架构。

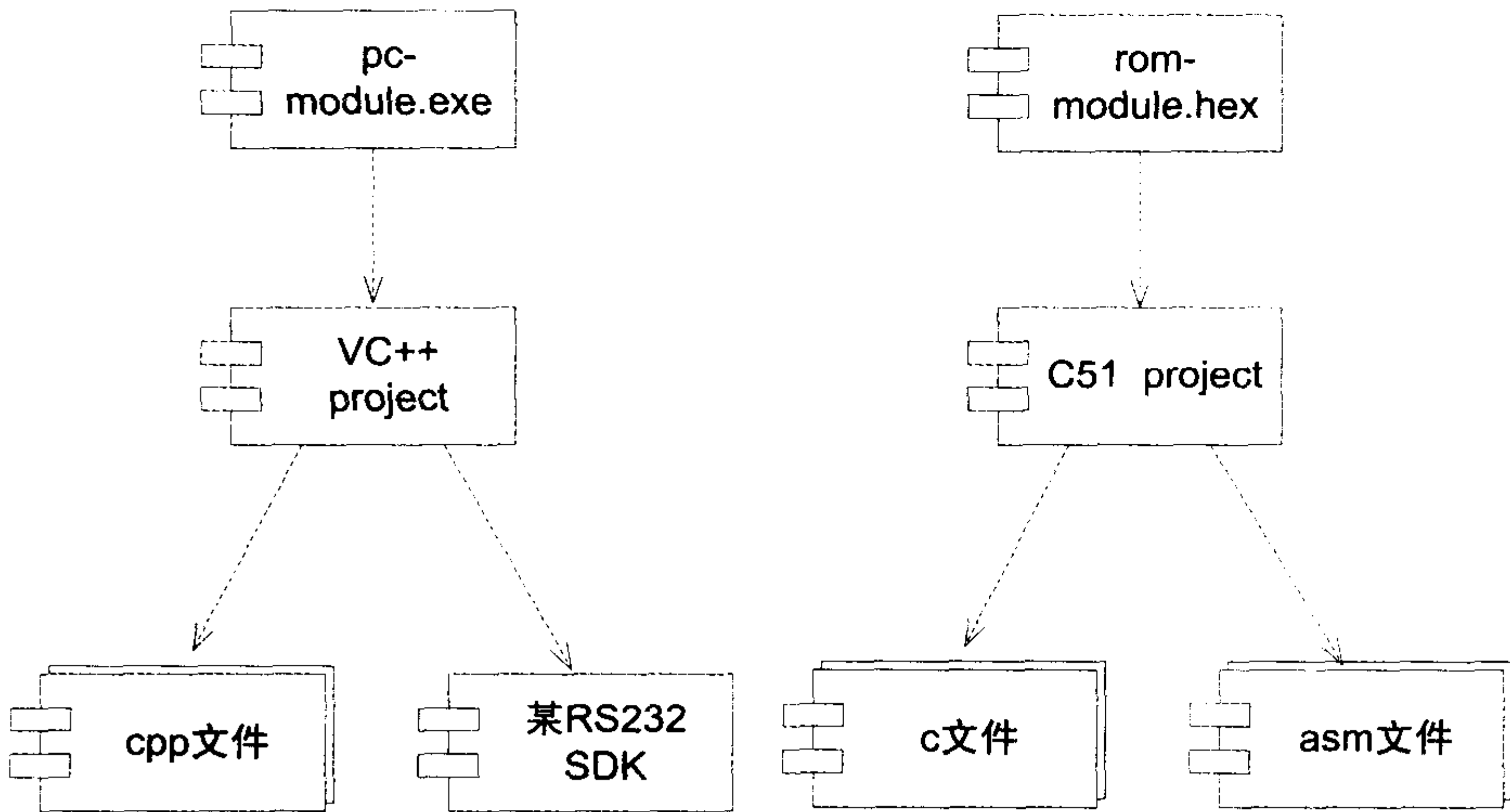


图 5-4 设备调试系统的开发架构

5.4.3 运行架构设计

性能是软件系统运行期间所表现出的一种质量水平，一般用系统响应时间和系统吞吐量来衡



量。为了达到高性能的要求，软件架构师应当针对软件的运行时情况进行分析与设计，这就是我们所谓的软件架构的运行架构的目标。运行架构关注进程、线程和对象等运行时概念，以及相关的并发、同步和通信等问题。图 5-5 展示了设备调试系统架构的运行架构。

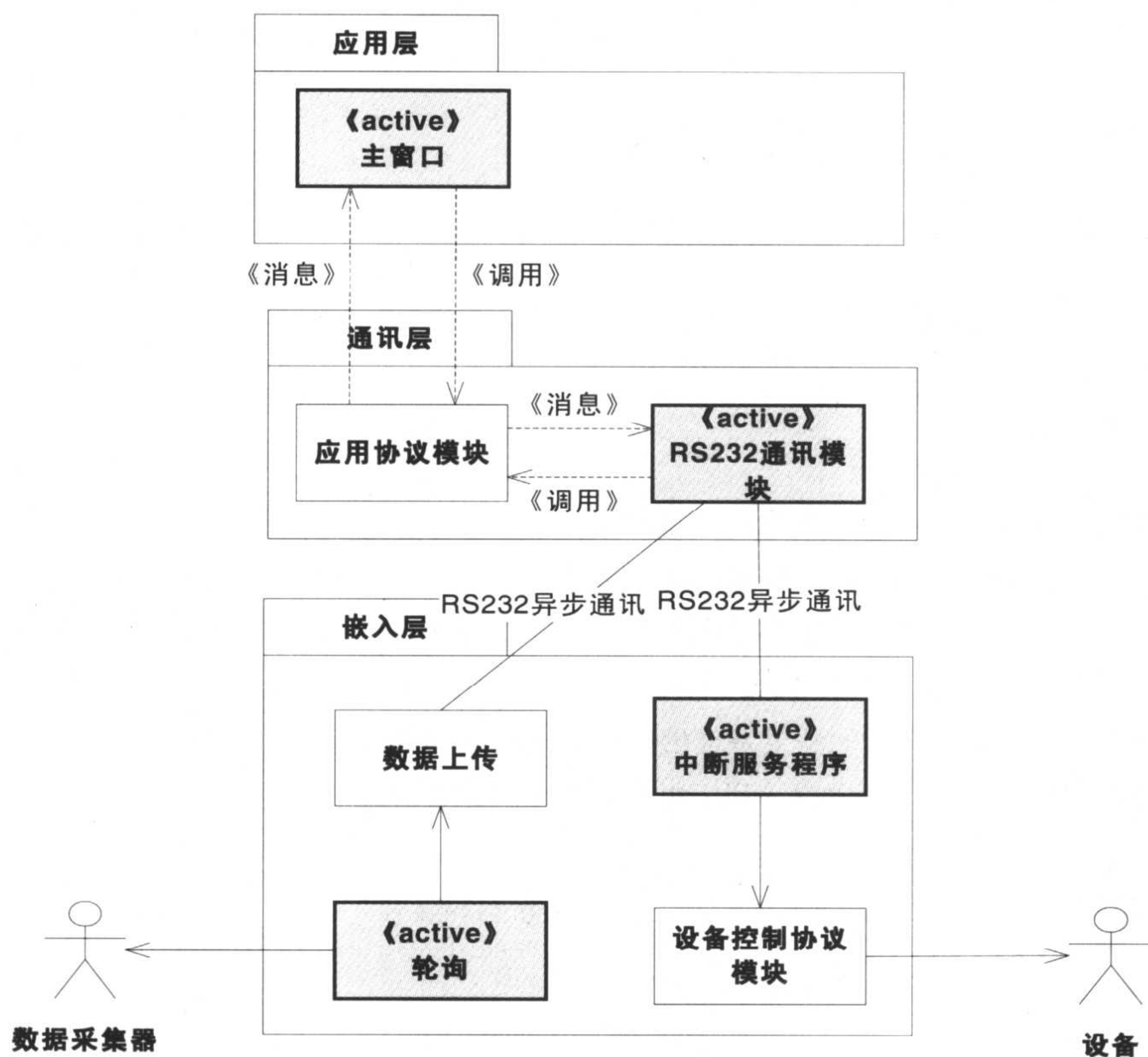


图 5-5 设备调试系统的运行架构

可以看出，架构师为了满足高性能需求，采用了多线程的设计：

- 应用层中的线程代表主程序的运行，它直接利用了 MFC 的主窗口线程。无论是用户交互，还是串口的数据到达，均采取异步事件的方式处理，杜绝了任何无谓的耗时（如“忙等待”等），也缩短了系统响应时间；
- 通讯层有独立的线程控制着“上上下下”的数据，并设置了数据缓冲区，使数据的接收和数据的处理相对独立，从而数据接收不会因暂时的处理忙碌而停滞，增加了系统的吞吐量；
- 嵌入层的设计中，分别通过时钟中断和 RS232 口中断来激发相应的处理逻辑，达到轮询和收发数据的目的。



---

## 5.5 总结与强调

---

方法如路标，为我们的实践提供很多指导。

本章讲述的架构设计的 5 视图方法包含下列 5 个架构视图：逻辑架构视图、开发架构视图、运行架构视图、物理架构视图、数据架构视图。构成每个架构设计视图的元素不同，这些不同的元素撑起了不同的思维空间，从而使每个架构视图重点覆盖不同种类的需求。最终，所有架构设计视图所表达的语义综合在一起，就构成了软件架构设计方案。

本书并不以“对比现有架构设计方法”为写作目的，但是，有熟悉 RUP 4+1 视图的读者可能希望了解本章的 5 视图方法和 4+1 视图方法有何不同。在此简要说明一下：

- 首先，我们去掉了“用例视图”。因为用例不足以驱动架构设计，例如软件系统的可扩展性等需求对架构设计很重要，但用例并不能表达此类需求，因为用例仅是围绕“能为用户带来可观察的结果的”动作序列展开的。本书专门讲述了如何为满足质量属性而设计，请参考第 15 章；
- 其次，我们增加了“数据视图”。这是根据企业应用的需要而增加的。



## 第 6 章 从概念性架构到实际架构

---

好的开始是成功的一半。

——谚语

少就是多 (Less is more.)。

——密斯·凡德罗

提供的细节与它的抽象层次一致。

——Grady Booch, 《UML 用户指南》

万事开头难。

当要设计的软件系统非常复杂时，直接设计实际架构往往有困难。而且，对于没有经验的软件架构师，基于多视图的架构设计方法或许还稍显复杂。

实际的软件架构设计过程是，一般应先进行概念性架构的设计，把最关键的设计要素和交互机制确定下来，然后再考虑具体技术的运用，设计出实际架构。

### 6.1 概念性架构

---

概念性架构是对系统设计的最初构想。概念性架构没有严格的定义，而且也不应该有过于严格的定义。为什么这么说呢？因为当解决任何复杂问题时，前期更重视找到解决办法，它往往是战略而不是战术，它比较策略化而未必全面，它比较强调重点机制的确定而不一定非常完整。

这就好比我们解数学题时，往往会首先“启发性地对解法进行计划”。

下面，给出概念性架构的一些描述。它们未必是定义，但对我们进行概念性架构设计很有启发：

- 概念性架构通过主要的设计元素及它们之间的关系来描述系统；



- 概念性架构符合“软件架构”的定义，从“架构=组件+交互”角度而言，概念性架构包含概念性组件以及它们之间的抽象交互机制；
- 概念性组件往往是粗粒度的（虽然从理论上抽象程度和粒度并无关系）；
- 概念性架构包括一些高层次的设计选择，对未来软件系统的质量和功能都起着关键影响；
- 概念性架构重在点明关键机制；
- 复杂系统的设计往往不能一蹴而就，而概念性架构就是最初的架构设计成果。

另外，隐喻和概念性架构有很大的关系。Kent Beck 在《解析极限编程》一书中写道：

..... 隐喻只是帮助项目中的每个人理解基本要素及其关系。

用于标识技术实体的词语应该从选择的隐喻中前后一致地提示。随着开发的继续进行和隐喻的逐渐成熟，整个团队将从隐喻的探讨中找到新的灵感。

XP 中的隐喻很大程度上取代了别人所说的“架构”。将那个 10000 米长的系统视图称作架构的问题是架构并不一定能够对系统进行任何意义上的浓缩。架构就是大方框和连接。

你可以说，“做得很差的架构当然很糟糕。”我们需要强调架构的目的，那就是向每一个人提供一个进行工作的连贯的情节，一个为业务和技术人员所周知的情节。通过引入隐喻，我们可以得到易于沟通和阐述的架构。

的确，设计概念性架构，隐喻是一个非常不错的手段。第 14 章详细讲述概念性架构设计时，我们会看到，很多架构模式其实都借助了隐喻的手段。

下面举例。

首先是分层架构的例子，它太流行了。图 6-1 所示为 MySQL 的概念性架构，从图中可以看出：

- 它明确了 MySQL 所包含的主要设计元素，包括查询引擎、事务控制器、缓冲区管理器、恢复管理器和存储管理等；
- 这些设计元素被分为三层（Layer）；
- 这些设计元素之间的依赖关系遵守分层架构模式的规定，即层间的依赖关系是单向的（此条必须），并且没有跨层依赖（此条可选）。

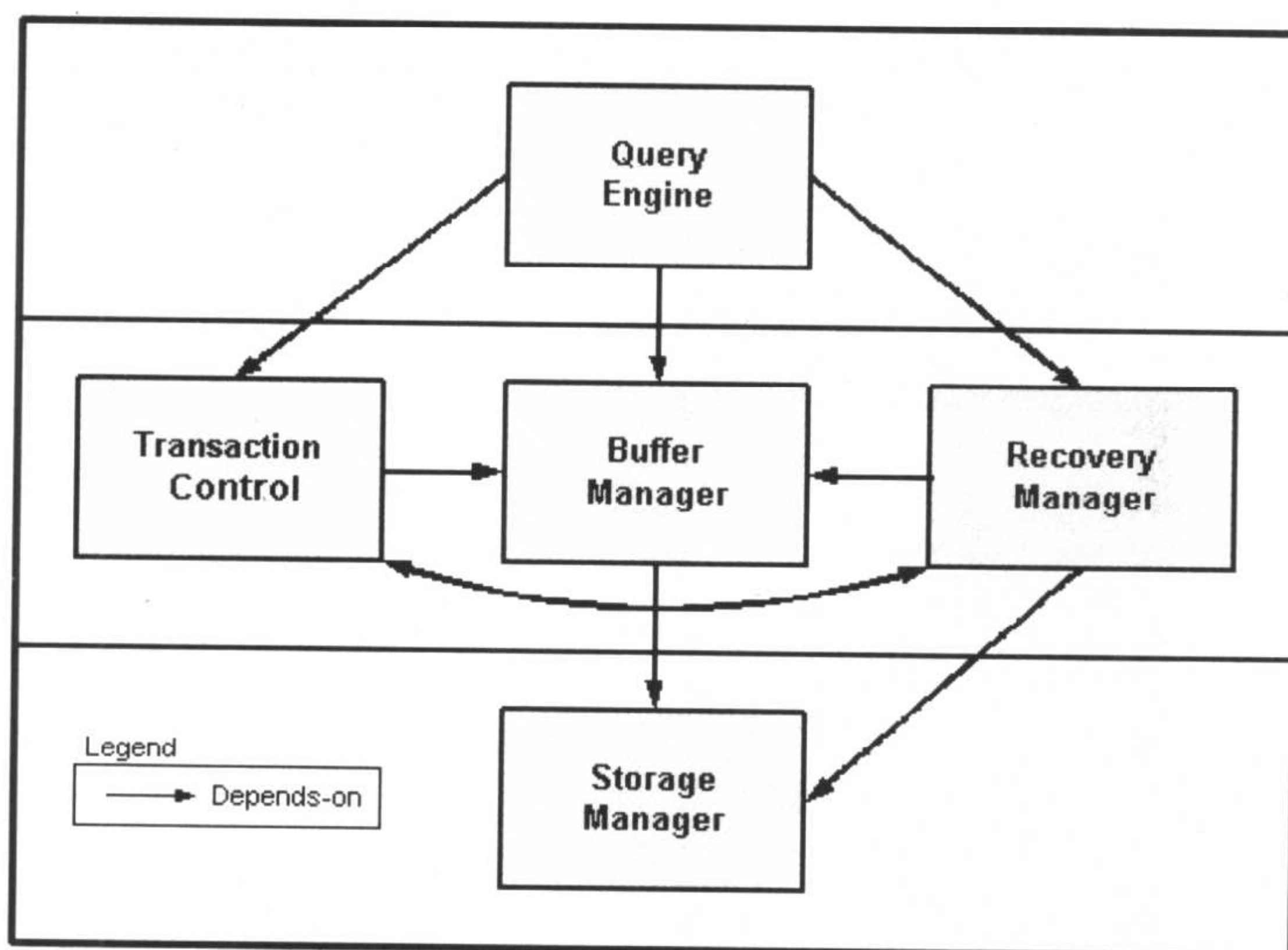


图 6-1 MySQL 的概念性架构 (图片来源: [http://www.swen.uwaterloo.ca/~rekram/files/cs798\\_assignment\\_1.htm](http://www.swen.uwaterloo.ca/~rekram/files/cs798_assignment_1.htm))

随着分布式应用的大行其道, 分层架构有了新的发展。经典的分层架构的层指逻辑层 (Layer), 而现在的分布式应用的分层架构也指物理层 (Tier)。逻辑层是一种功能和职责的组织单元, 而物理层是一种支持分布式部署的组织单元; 其实, 物理层也要求功能和职责的高聚合性, 只不过往往是通过多个逻辑层映射到一个物理层这种方式实现的。仅举一例, 图 6-2 所示为 J2EE 的概念性架构, 它更关注可分布式部署的物理层概念及它们之间的交互关系。

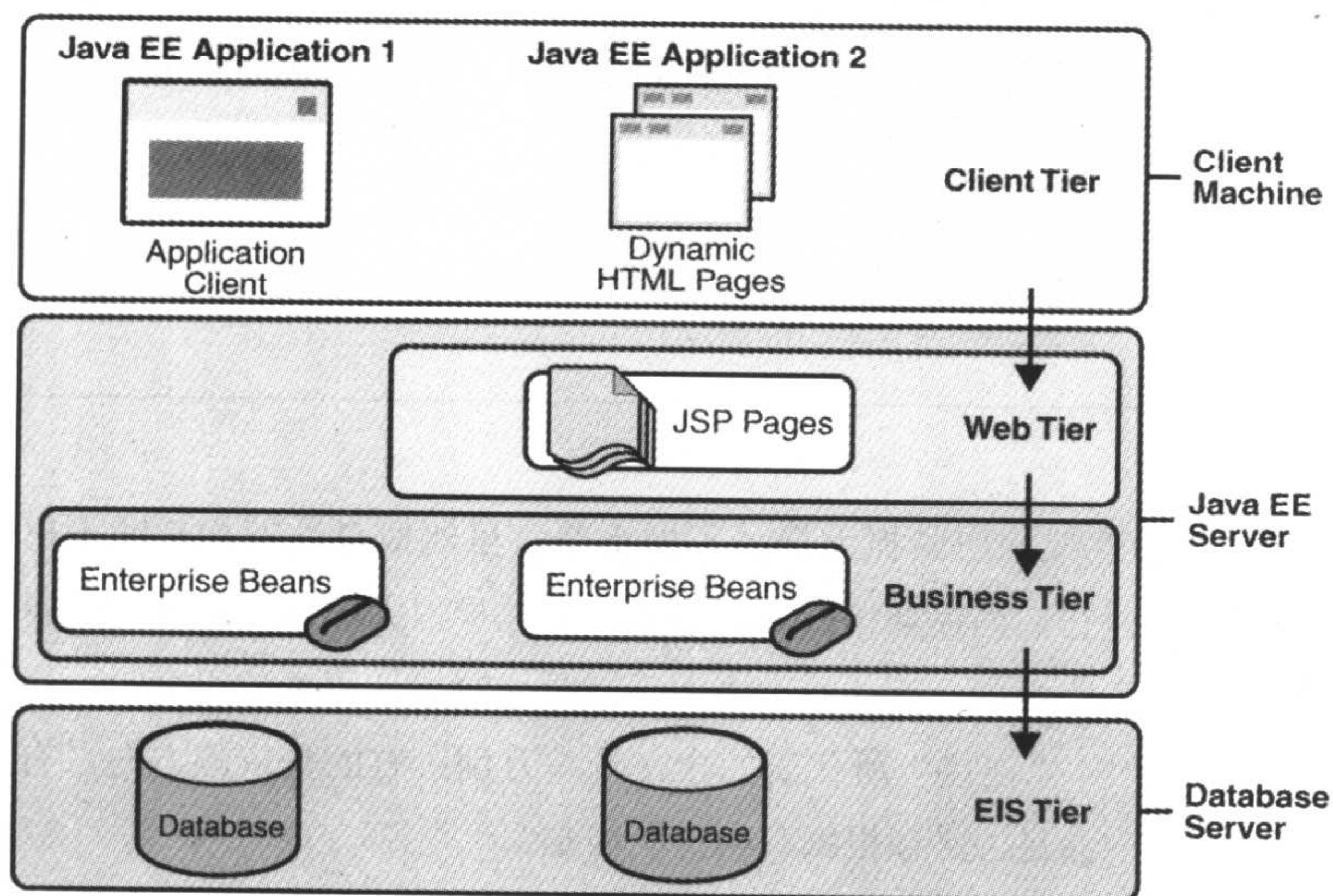


图 6-2 J2EE 的概念性架构 (图片来源: 《The Java EE 5 Tutorial》)



再看看 JBoss 的架构，如图 6-3 所示。这当然是概念性架构，虽然明确了采用微内核架构模式，但还有非常多的架构设计工作要做。

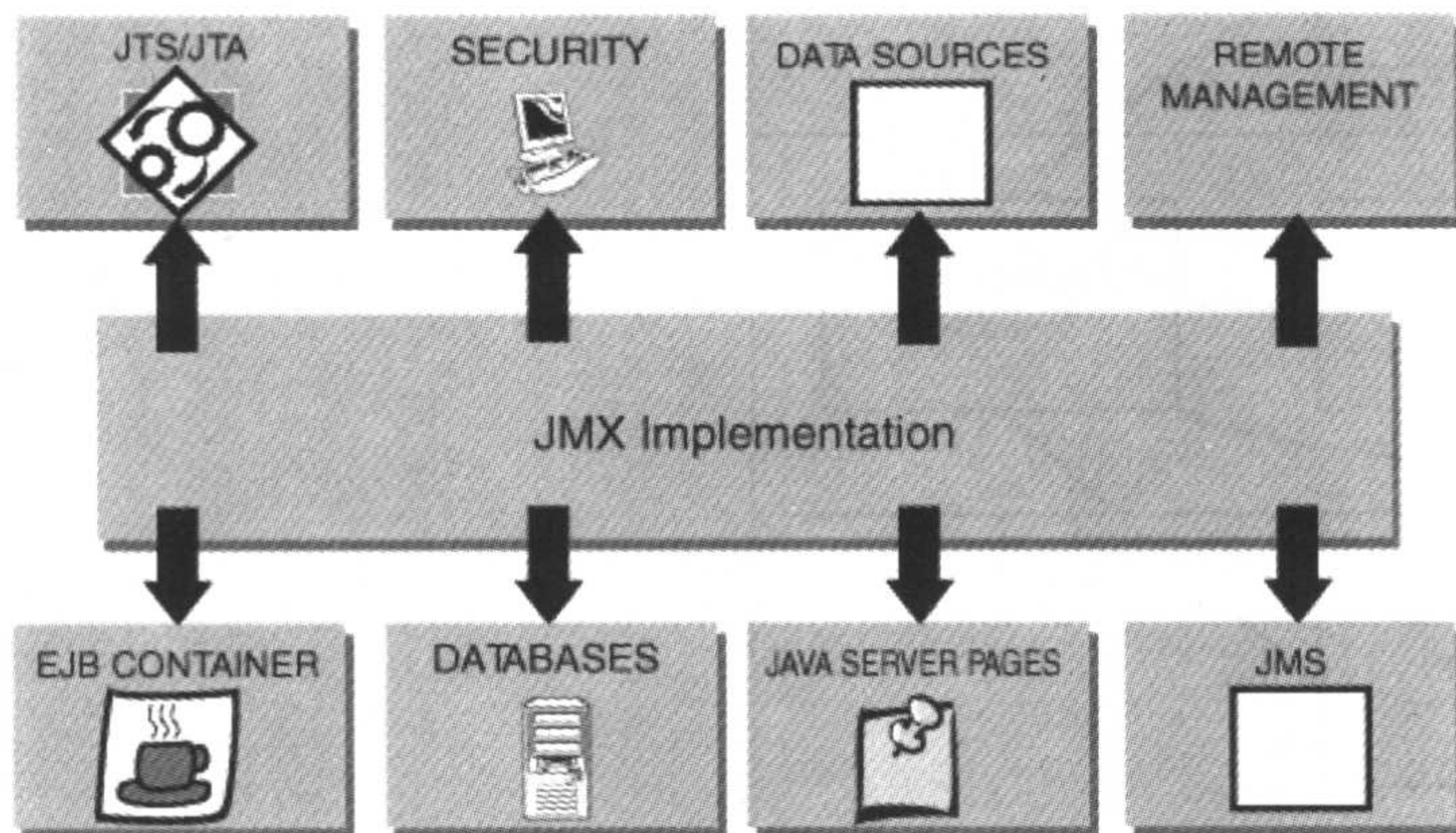


图 6-3 JBoss 的概念性架构（图片来源：JBoss 官方文档）

可以举的例子很多，例如 SOA（Service-Oriented Architecture，面向服务架构）、MDA（Model Driven Architecture，模型驱动架构）和 CBD（Component Based Development，基于组件的开发）等的概念性架构都可圈可点。图 6-4 展示了 Web 服务的概念性架构。它很典型，有很重的“隐喻”气息：服务请求者、服务代理者和提供者很形象地说明了它们各自的角色，再加上发布、寻找和绑定这些交互关系的隐喻描述，Web 服务的核心机制跃然纸上。

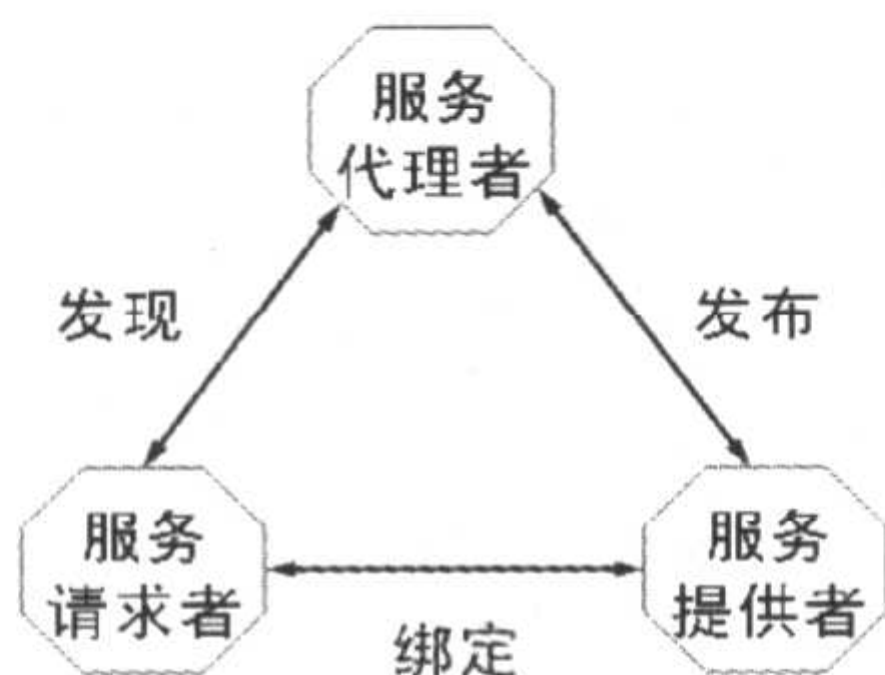


图 6-4 Web 服务的概念架构

通过上面的讨论可以看出，概念性架构应紧抓大局、不拘小节。虽然概念性架构都跳不出“架构=组件+交互”的基本定义，但它们描述架构的具体方式还是差异很大的：有的重视逻辑层，有的重视物理层，有的通过隐喻表明机制，有的看上去似乎就是一些设计元素的组合，……最为明显的是，不同的概念性架构图中，“连接”所代表的含义千差万别：有的是依赖方向，有的是控制方向，有的是数据流向，不一而足，必须根据具体情况而定。



## 6.2 实际架构

由于概念性架构的高度抽象性，使得同属一类的许多软件产品的概念性架构是趋同的。

例如，人们已开发了大量的 J2EE 应用，它们的概念性架构很多都是如图 6-2 所示的架构。但是，它们的质量为什么可能判若云泥？

再例如，在很大程度上，如果图 6-1 所示的 MySQL 的概念性架构，也可以说是 Oracle 的概念性架构。但是，MySQL 和 Oracle 在一些方面的差异为什么却那么大呢？

答案在于，即使概念性架构非常相似，实际架构却可能有很大差异。图 6-5 说明了实际架构和概念性架构的不同。概念性架构往往和具体技术的运用、具体平台的选择无关，而实际架构则非常关心这些问题。

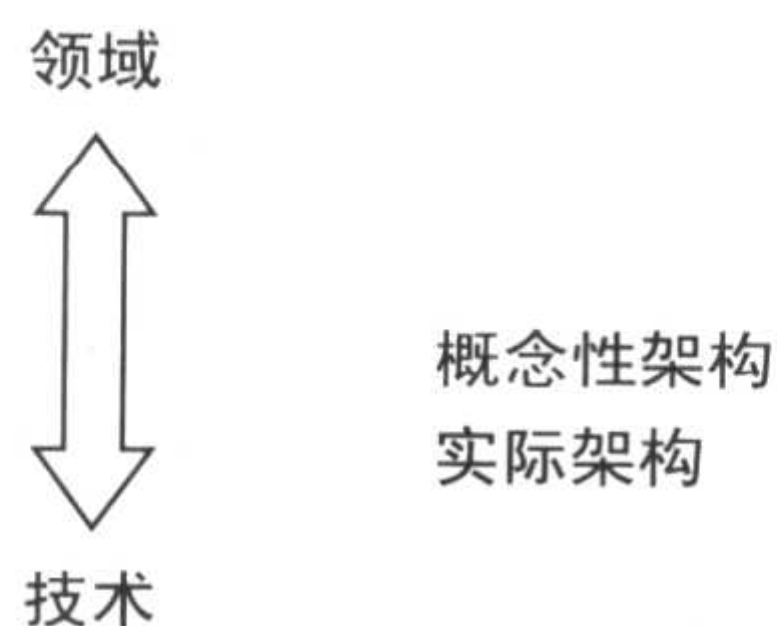


图 6-5 实际架构和领域架构的“位置”

如前所述，实际的架构设计方案应该兼顾不同的架构设计视图。那么，概念性架构又与多个架构视图是什么关系呢？

一般而言，概念性架构所包含的高层设计决策终究不会跳出如下多种架构视图的范围——逻辑架构、物理架构、开发架构、数据架构或运行架构。只不过，概念性架构设计的抽象程度比较高，设计程度（图中的阴影代表工作量）也很不充分，而实际架构必须设计到可以指导开发的程度。图 6-6 说明了这一点。

但务实地来讲，概念性架构经常从逻辑架构和物理架构的角度制定高层设计决策；例如，图 6-1 的 MySQL 概念性架构偏重逻辑架构视图，而图 6-2 的 J2EE 概念性架构偏重物理架构视图。

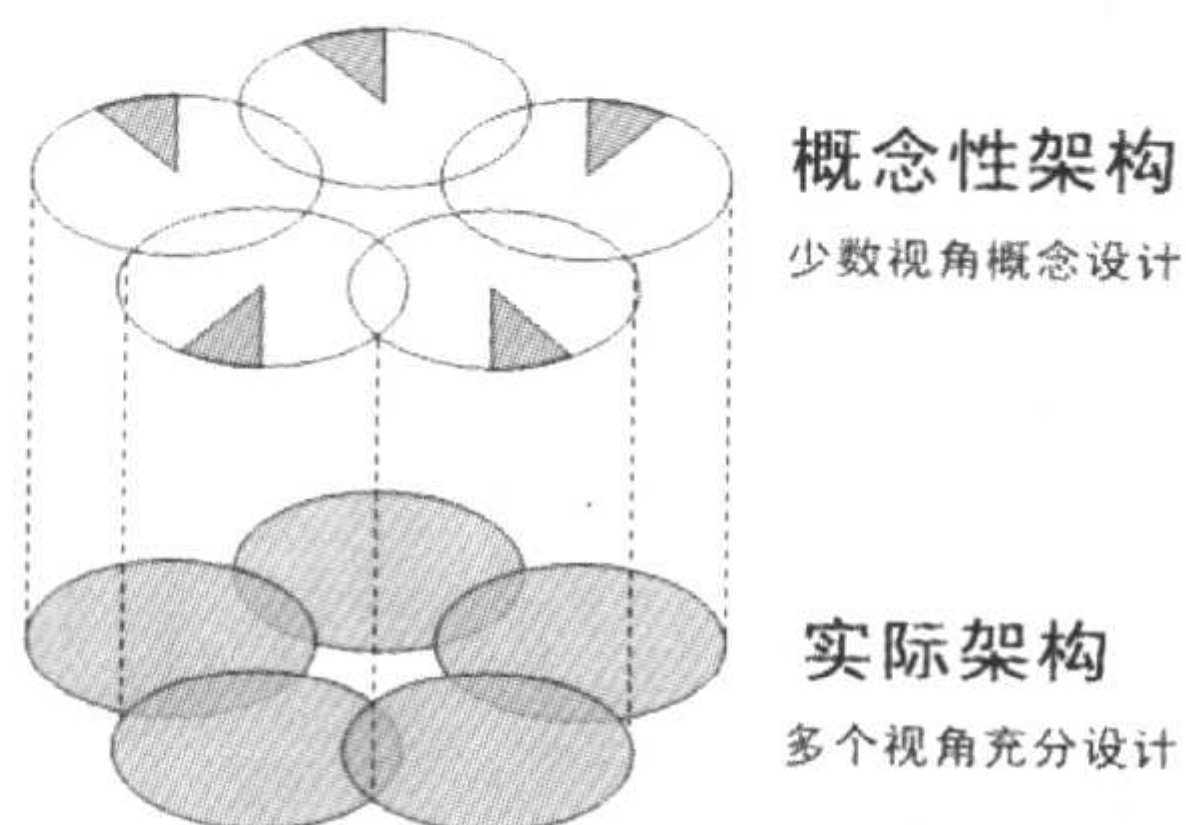


图 6-6 概念性架构与多重架构视图的关系

## 6.3 从概念性架构到实际架构

概念性架构是不可直接实现的。开发人员拿到概念性架构设计方案，依然无法开始具体的开发工作。从概念性架构到实际架构，要运用很多具体的设计技术，开发出能够为具体开发提供更多指导和限制的实际架构。

下面是概念性架构和实际架构的一些区别，涉及了开发人员最为关心的“点”：

- 接口。在实际架构中，接口占据非常核心的地位；而概念性架构中没有接口的概念（只有抽象的组件和抽象的交互机制）；
- 子系统。实际架构重视通过子系统和模块来分割整个系统，并且子系统往往有明确的接口；而概念性架构中只有抽象的组件，这些组件没有接口只有职责，一般是处理组件、数据组件或连接组件中的一种。当然，概念性架构中也有“大组件分解成小组件”这样的设计决策，但并非子系统的含义；
- 交互机制。实际架构中的交互机制应是“实在”的，如基于接口编程、消息机制或远程方法调用等等；而概念性架构中的交互机制是“概念化”的，例如“A层使用B层的服务”就是典型的例子，这里的“使用”到了实际架构中可能是基于接口编程、消息机制或远程方法调用等其中的任一种；
- 共同点。概念性架构和实际架构都满足软件架构的概念——无论是“架构=组件+交互”，还是“架构=重要决策集”。

## 6.4 网络管理系统案例：从分层架构开始

分层架构作为最为流行的架构模式，时常出现在很多系统的概念性架构中。更为重要的是，分层架构往往是我们架构思维的开始——换言之，在“分层”的基础上如何深入（细化设计）、如何调整（可能最终设计和经典的分层有较大差异）往往是架构设计的关键。

本节的写作要达到什么目的？概括如下：

- 通过案例分析，使读者感受到概念性架构和实际架构之间的“跨度”；
- 暗示分层之后就止步不前，这样的架构设计是不充分的；
- 说明OO技术在“支撑”概念性架构和实现实际架构中的作用。

### 6.4.1 构思：概念性架构设计

开始，你或许只有一个朦胧的想法：三层架构（如图6-7所示）。



图 6-7 三层架构

你或许还会极度困惑：到底采用分层架构还是 MVC 架构呢？这是个“流行的困惑”，但其实：

- 多种架构模式可以联合使用，它们或者“作用于”系统的不同部分，或者“作用于”不同的级别（例如总体架构为 MVC 而控制层本身由于复杂又采用分层架构来组织）；
- 从理论上，MVC 架构是分层架构的一种特殊情况；而在实践中，开始构思分层架构，后来明确交互机制后就“变成” MVC 架构的情况也时有发生。

回到网络管理系统的例子。为了保证网络持续、稳定、安全和高效地运行，网管软件的业务层应提供丰富的功能，如配置管理、故障管理、安全管理和性能管理等，因此业务层会包含一个比较复杂的模型；而展现层应以业务层的模型为基础，提供各种展现视图、控制界面和查询方式等。因此，展现层和业务层之间应以 MVC 模式规定的交互机制为主。这样一来，就出现了我们上述的“架构模式联用”的情形，如图 6-8 所示。

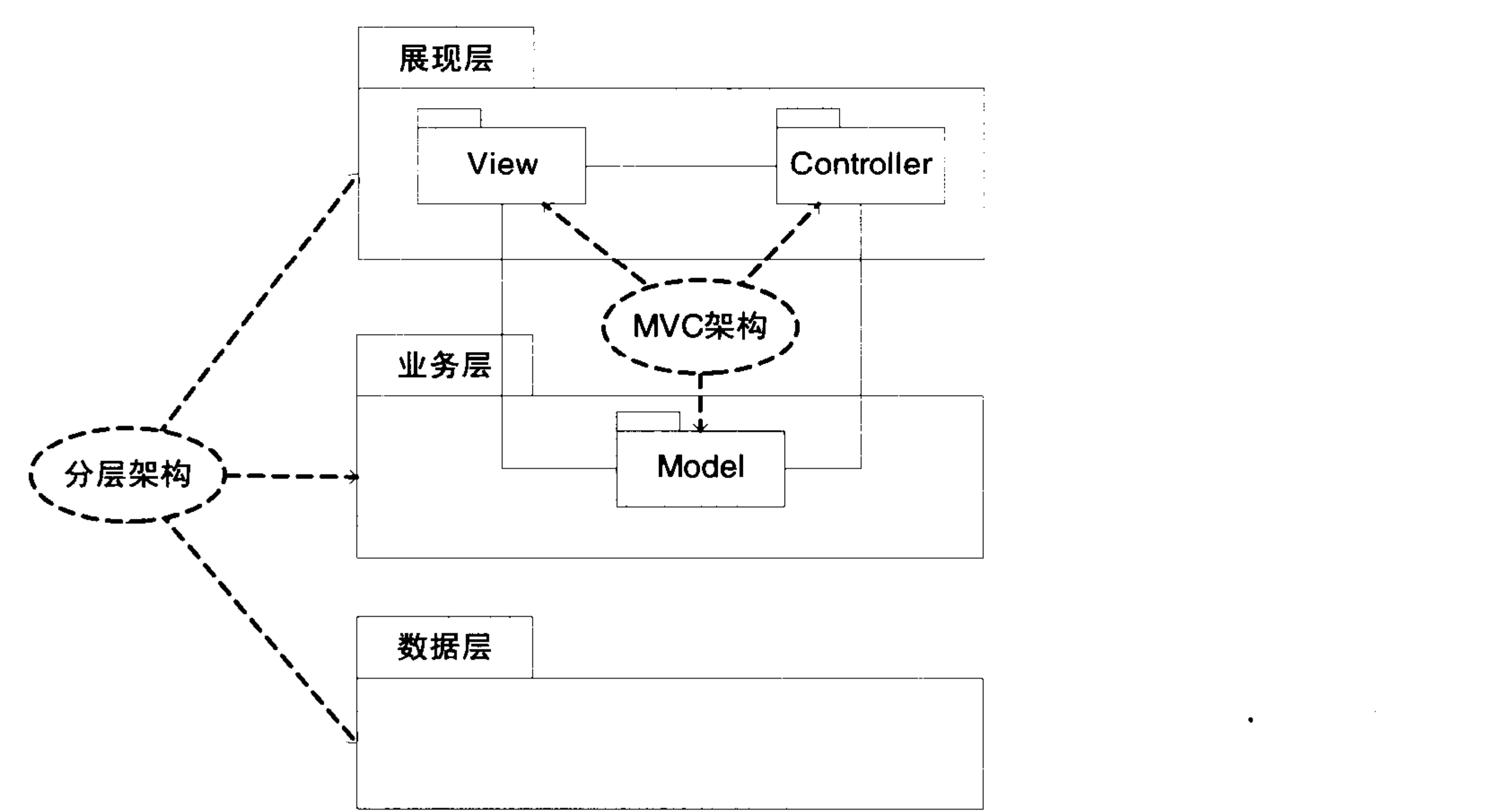


图 6-8 在分层的基础上，进一步引入 MVC 架构



概念性架构必须和公司或组织的业务情况相符合。但当前这个设计却做不到这一点。这是因为，我们这个案例中的“主角”作为专业从事网络管理系统开发的软件公司，不是仅有一个网管产品，而是拥有完整的网管软件产品线——产品线中不同软件产品的功能既有共性又有不同，它们往往建立在一组可重用的公共软件包之上。

必须进行产品线架构的设计才符合公司的业务情况！图 6-9 展示了“最后的”分层架构——网管系统产品线的概念性架构。

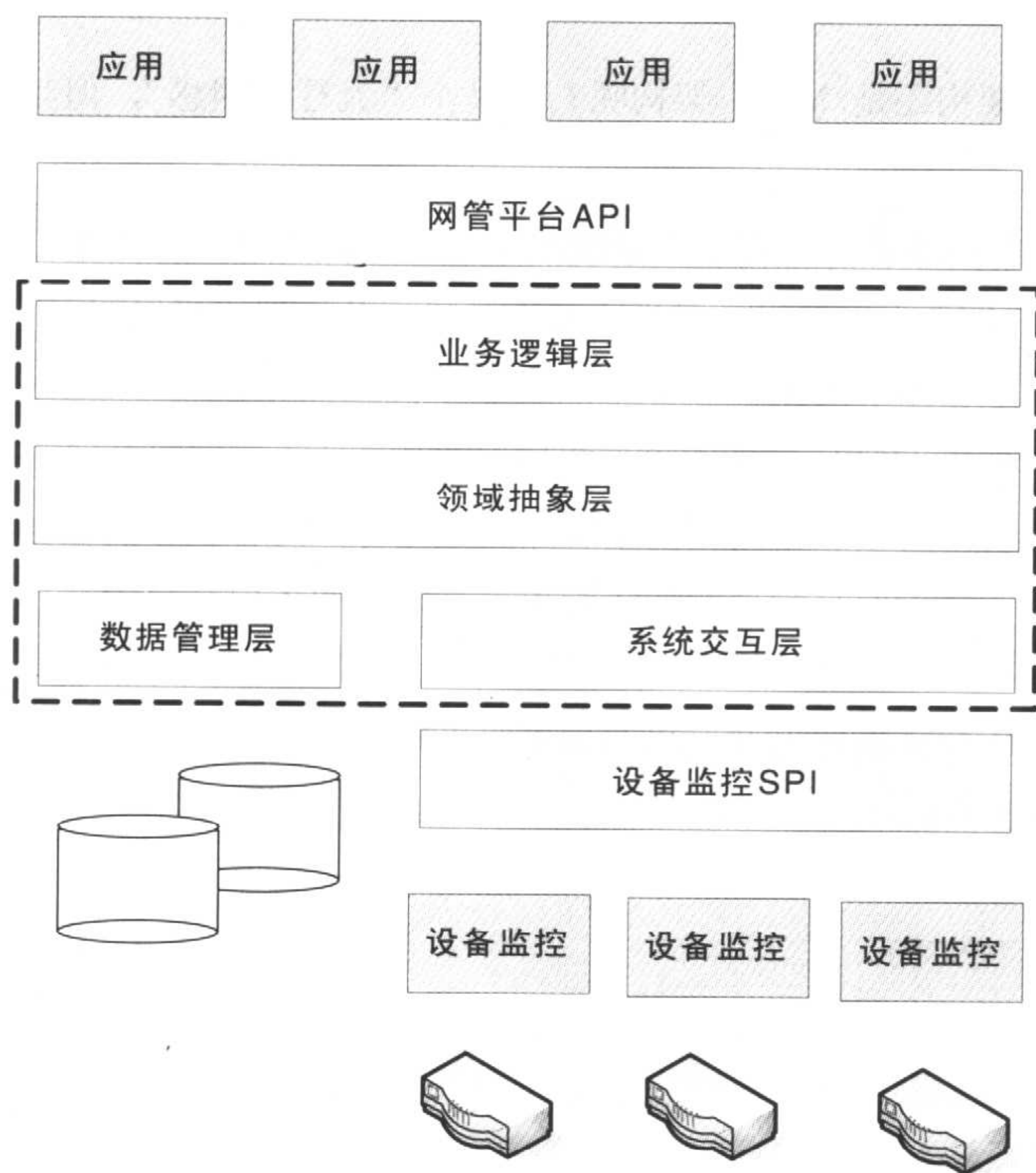


图 6-9 网管系统产品线的概念性架构

这个概念性架构充分考虑了未来可能发生的变化，例如：

- 应用的具体形式是丰富多彩的，而产品线架构提供了和具体平台无关的“网管平台API”，从而可以方便地支持桌面应用、Web 应用和插件式应用（如 Eclipse 插件）等形式的应用；
- 将网络设备的监控部分分离出去，但为它们制定统一的服务编程接口（SPI，Service Programming Interface），从而达到“隔离变化”的效果。这样做的好处很多：一是便

于分布部署设备的监控服务部分，从而提高性能和方便管理；二是下游合作伙伴可以自行扩充网管系统的能力，并且不会有版权问题。

6.4.2 深入：实际架构设计

如前所述，概念性架构往往和具体技术的运用和具体平台的选择无关，而实际架构则非常关心这些问题。例如，OO 技术就是“具体技术”的典型例子：概念性架构并不规定要采用 OO 技术，但设计实际架构时往往要考虑如何充分利用 OO 技术来实现概念性架构。

不难看出，网管平台“上”有标准 API、“下”有标准 SPI，这一设计很关键。下面讨论如何运用 OO 技术实现 API 机制。

第一步，运用 OO 技术对实现 API 和 SPI 进行了初步规划。如图 6-10 所示，我们决定运用“面向接口编程”：“网管平台 API”和“设备监控 SPI”这两个包主要包含一些接口定义，通过这些接口可以将变化隔离。

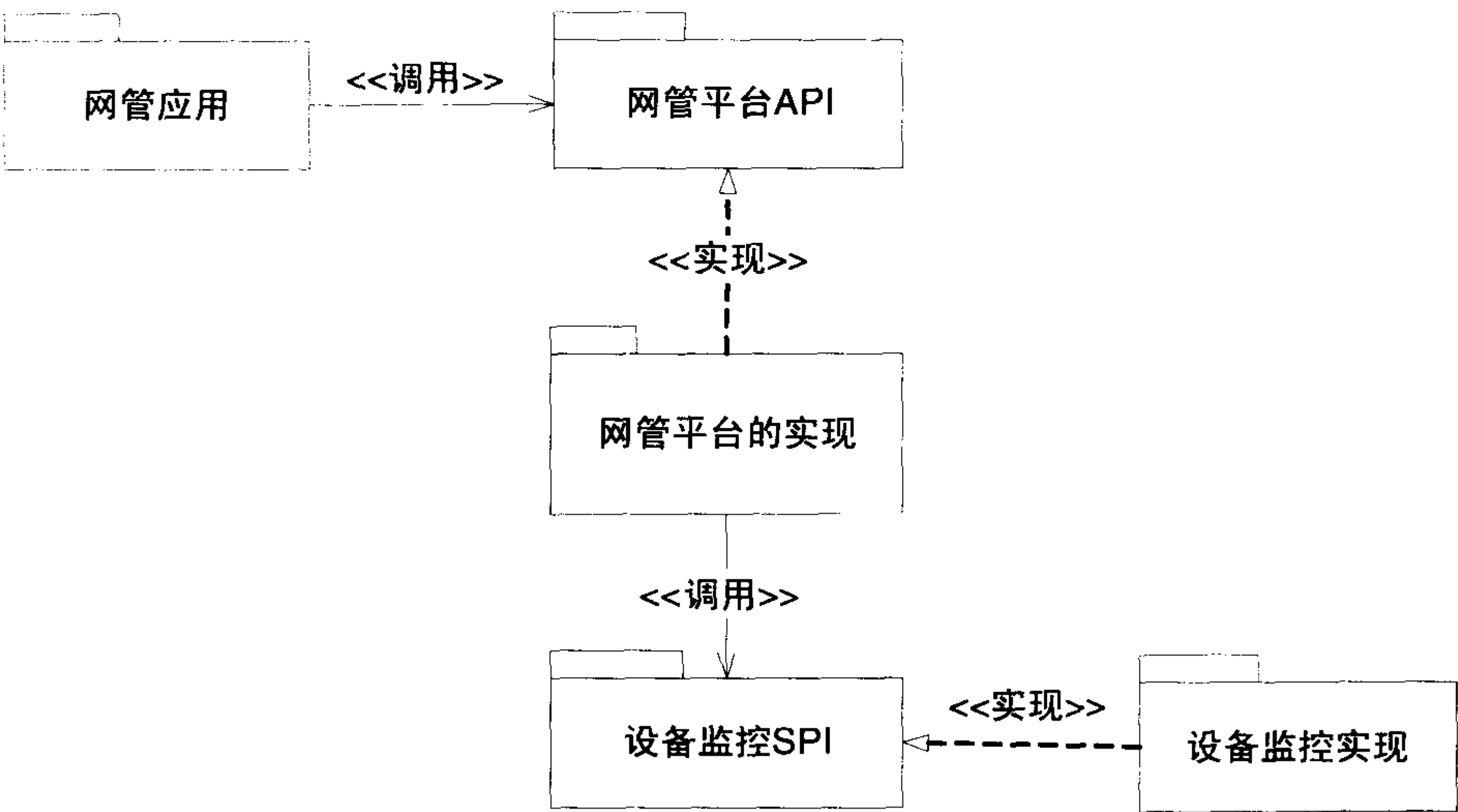


图 6-10 引入 OO 的第一步

图 6-11 展示了网管平台 API 的一部分以及它是如何隔离变化的。

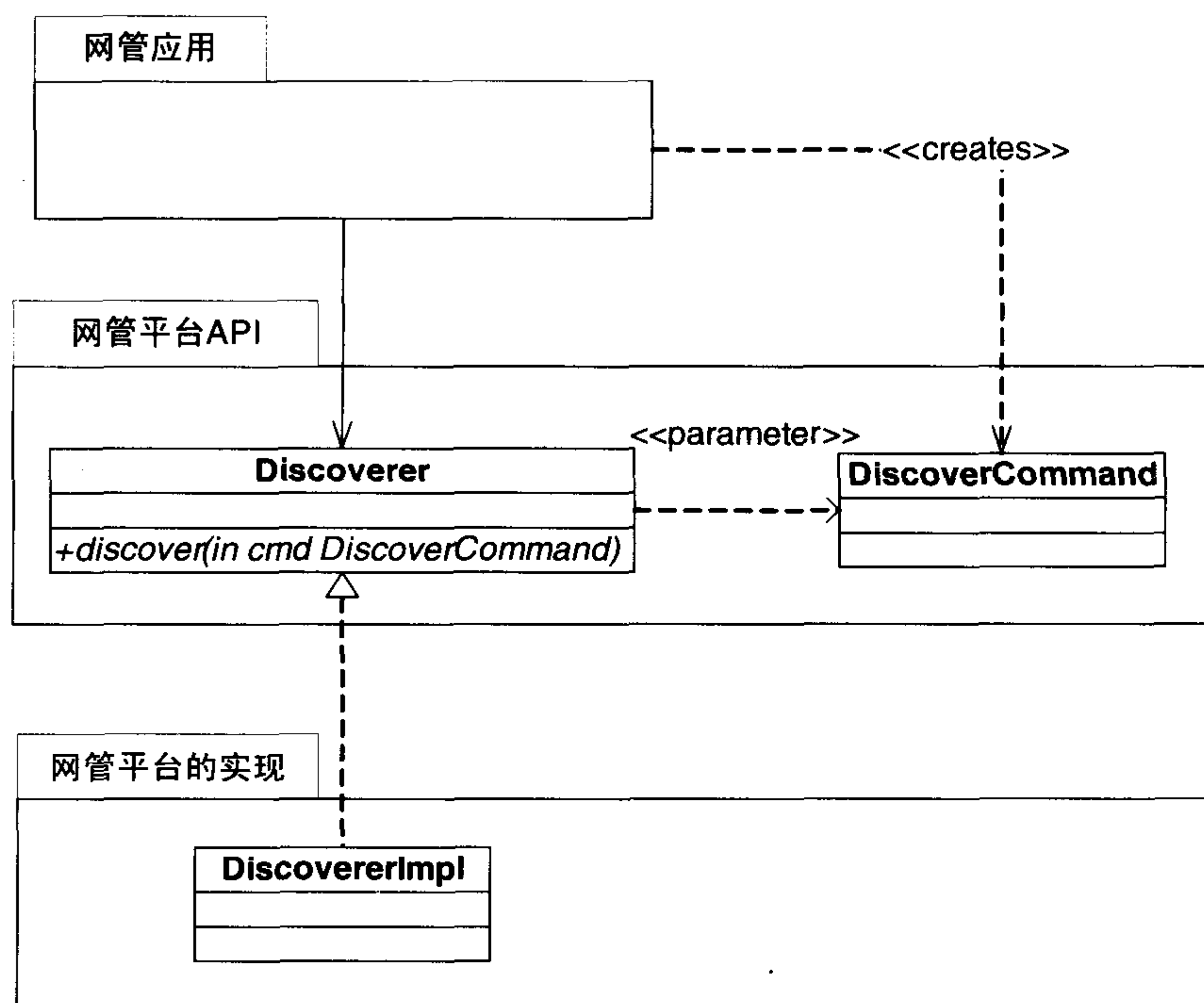


图 6-11 实际架构设计的一部分

## 6.5 总结与强调

从概念性架构到实际架构，先设计概念性架构，构思关键问题的解决策略；再进行实际架构的设计，以保证为开发提供足够的指导和限制……这符合人类解决问题的规律，因此被广泛采用。

本章着重讲述概念性架构和实际架构的不同，并通过网络管理系统案例的分析说明了如何从概念性架构设计过渡到实际架构的设计。读者可以通过第 14 章和第 16 章更详细地了解开展概念性架构设计和实际架构设计的具体方法。



## 第 7 章 如何进行成功的架构设计

---

策略是制胜的关键。

——张明正，《挡不住的趋势》

人类知识和人类权力归于一；因为凡不知原因时即不能产生结果。……  
而凡在思辨中为原因者在动作中则为法则。

——培根，《新工具》

最好的软件开发人员都知道一个秘密：美的东西比丑的东西创建起来更  
廉价，也更快捷。

——Robert C. Martin，《软件之美》

一个软件项目是否能够成功，软件架构十分关键。

仔细观察我们所处的软件业界，许多“软件项目”因为架构问题而返工甚至失败，还有许多“软件产品”因为架构问题而不得不重新开发新的版本。

那么，如何使软件架构设计工作取得成功呢？在看似相同的外表之下，成功的架构设计和失败的架构设计有何关键性的不同呢？本章从探究成功架构设计的关键要素入手，循序渐进地展开分析；之后，总结 4 条对架构设计成功至关重要的架构设计策略，并对每一条策略进行讲解。

### 7.1 何谓成功的软件架构设计

---

所谓成功的架构设计，就是设计出的软件架构是高质量的，并且在所花费的时间、技术决策等方面也都满足具体开发情况的要求。

好的软件架构应当具有如下品质：

- 良好的模块化。每个模块职责明晰，模块之间松耦合，模块内部高聚合并合理地实现了信息隐藏；
- 适应功能需求的变化，适应技术的变化。典型地，应该保持应用相关模块和领域通用

模块的分离，技术平台相关模块和独立于具体技术的模块相分离，从而达到“隔离变化”的效果；

- 对系统的动态运行有良好的规划。标识出哪些是主动模块，哪些是被动模块——面向对象中往往是主动类（Active Class）和被动类（Passive Class），明确这些模块之间的调用关系和加锁策略，并说明关键的进程、线程、排队、消息等机制；
- 对数据的良好规划。不仅应包括数据的持久化存储方案，还可能包括数据传递、数据复制和数据同步等策略；
- 明确、灵活的部署规划。还往往涉及到可移植性、可伸缩性、持续可用性和互操作性等大型企业软件特别关注的质量属性的架构策略。

面对时间紧迫的压力，我们有理由质疑那种不顾时间花销、一味追求软件架构高质量的作法。软件架构是软件系统质量的核心，必须足够重视，但在不适当的时候“用时间换完美”会毁掉整个项目。我们可以反过来思考这个问题：之所以要在绝大部分技术细节都不清楚的情况下定义出软件架构，除了可以对性能、稳定性等系统整体质量属性进行综合考虑以外，还有非常重要的一点就是要搭建一个团队协作开发的基础，让不同小组分头对不同的系统模块深入下去，团队并行工作最终意味着缩短了项目工期。对此，Philippe Kruchten 曾一针见血地指出：“时间就是系统架构设计的生命。”

同时，架构设计并非“好的就是成功的”，而是“适合的才是成功的”。在架构设计过程中，架构师没有绝对的技术选择的自由，而是要充分考虑经济性、技术复杂性、发展趋势和团队水平等多方面的因素，制定出合适的架构决策。最终，软件架构师的工作成果要为整个软件开发团队的工作提供足够的指导和限制，使他们能够沿着正确的方向进行下去。

## 7.2 探究成功架构设计的关键要素

看似没什么不同。软件架构师开展架构设计工作，都是以《软件需求规格说明书》为最主要的设计依据的，都是先勾画出概念性架构，再结合具体技术平台制定实际架构的。

其实不然。中国有“外行看热闹，内行看门道”的说法，架构设计工作之中就藏着很多“门道”——即成功的关键要素。

### 7.2.1 是否遗漏了至关重要的非功能需求

为了使软件架构设计获得成功，我们必须在架构设计工作中注意几个关键要素，下面一一讨论它们。第一个关键要素是：你是否关注了至关重要的非功能需求。

**非功能需求来自何处？**

一部分非功能需求来自用户。用户要功能，用户也要质量。诸如性能、易用性等软件质量属

性，虽然不像功能需求那样直接帮助用户达到特定目标，但并不意味着软件质量属性不是必需的——恰恰相反，质量属性差的软件系统大多都不会成功。例如，你提供了用户要求的“交易查询”功能，但这一功能动辄就需要花上几分钟，用户能接受吗？当然不能接受。用户会说，功能虽然具备了，但质量太差难以接受。用户在使用软件系统的过程中，其关心的质量属性可能包括易用性、性能、可伸缩性、持续可用性和鲁棒性等。因此，软件架构师也应当时时牢记：为用户而设计，不仅要满足用户要求的功能，也要达到用户期望的质量。

一部分非功能需求来自开发者和升级维护人员。软件的可扩展性、可重用性、可移植性、易理解性和易测试性等非功能需求，都属于“软件开发期质量属性”之列，它们都将深刻影响开发者和升级维护人员的工作，这方面拙劣的质量会使开发和维护变得困难，时间和金钱方面的成本都会增加。例如，一个预计服务 5~8 年的软件系统，由于可扩展性拙劣而造成只使用了 3 年就不得不重新开发新系统，这在一定程度上也是失败的，因为投资没有支持足够的年限，就像我们买了一台电视机没多久就不能用了一样。

还有一部分非功能需求来自客户组织。架构师必须充分考虑客户对上线时间的要求、预算限制以及集成需要等非功能需求，还要特别关注客户所在领域的业务规则和业务限制。架构师应当直接或（通过系统分析员）间接地了解 and 掌握上述需求和约束，并深刻理解它们对架构的影响，只有这样才能设计出合适的软件架构。合适的才是最好的，例如，如果客户是一家小型超市，软件和硬件采购的预算都很有限，那么你就不宜采用依赖太多昂贵中间件的软件架构设计方案。

### 非功能需求为何对架构设计如此重要？

非功能需求是最重要的“架构决定因素”之一。非功能需求大致分为质量属性和约束两大类。质量属性是软件系统的整体质量品质——所谓整体品质，就是它往往和大多数功能都有关，而不是仅仅表现在某个功能“内部”。易用性、性能、可伸缩性、持续可用性、鲁棒性、安全性、可扩展性、可重用性、可移植性、易理解性和易测试性等都可能是软件的质量属性需求。当然，这些质量属性之间存在一定的相互矛盾的情况，因此实际上只有少数几个质量属性在架构设计中的重要性最高，它们通常会左右架构风格的选择。至于约束性需求，它们要么是架构设计中必须遵循的限制（例如“将运行于 Linux 平台”），要么转化为质量属性需求或者功能需求（例如“某系统的目标用户的计算机水平不高”实际上要求系统应有较高的易用性）。

反过来，最终提供的软件系统是否能满足非功能需求的要求，仅凭编程级的努力是达不到的。例如，为了使软件系统提供 7×24 小时的高持续可用性（High Availability），需要采取的架构设计策略可能涉及错误检测、修复和预防等。正因为如此，软件架构设计过程中必须重视非功能需求。

总之，功能很重要，但架构师不能仅盯着功能需求，若忽视了非功能需求，则可能导致架构设计的失败。



### 7.2.2 能否驯服数量巨大且频繁变化的需求

需求的变更可以说“让我欢喜让我忧”——其中既蕴藏了风险又包含了机遇。

之所以说需求变更蕴含着风险，是因为不存在不需要成本的需求变更。任何需求变更都可能意味着时间和金钱的消耗，并且大量需求变更之后程序可能被搞得混乱不堪，势必引起 Bug 增多，影响产品质量。

当然，需求变更也蕴含着机遇。对软件架构设计而言，这个机遇可能意味着设计出稳定的架构，最终这个架构能够支持业务功能在一定范围内“按需应变”。

从深层次来讲，新经济最具挑战的一个特点就是频繁变化性。纵观商业企业的“生态环境”，未来将越来越不可预测，商业企业“按需应变”的要求日趋显著。另一方面，技术也不是单纯处于为商业服务的位置，相反，技术在一定程度上会拉动需求并促进需求变化；除了互联网技术这一最明显的例子之外，我早年读《只有偏执狂才能生存》时也深感 Wintel 联盟在“创造需求”方面的高明；例子还有很多……这些都是软件需求变更压力的深刻根源。

不驯服数量巨大且频繁变化的需求，软件架构师就难以“从容”地进行架构设计，最终甚至被变化拖垮而无法关注大局。

### 7.2.3 能否从容设计软件架构的不同方面

架构师必须具备“忘却”的能力，避免涉及太多具体的技术细节，但是软件架构可能依然是复杂的。

软件架构必须为开发提供足够的指导和限制，这在软件系统越来越复杂的今天，无疑意味着软件架构是复杂的。例如，为了满足性能、持续可用性等方面的需求，架构师必须深入研究软件系统运行期间的情况，合理划分系统不同部分的职责，权衡轻重缓急，并制定相应的并行、分时、排队、缓存和批处理等设计决策。而为了满足可扩展性、可重用性等方面的需求，则要求架构师深入研究软件系统开发期间的职责划分、变化隔离、框架使用和代码组织等情况，制定相应的设计决策。

因此，软件架构师需要掌握趋于系统化的方法。基于多视图的架构设计方法就是这样一种方法，有助于架构师设计出满足不同涉众需求的软件架构。有人说，“方法产生于恐惧”；我想，说“方法产生于复杂”也是合适的。

对待复杂性的办法就是分而治之，这一点似乎从来就没有改变过。需要说明的是，和分而治之相伴相生的“综合考虑”也是不可或缺的——对于多视图的架构设计方法来说，多个视图之间的相互支持、相互影响往往需要“综合考虑”。

### 7.2.4 是否及早验证架构方案并作出了调整

现代软件开发非常注重尽早降低风险。所谓风险，就是如果你忽略了它的存在，它就会主动找上门儿来的那种家伙。因此，我们必须主动防范风险。

架构设计是现代软件开发中最为关键的一环，架构设计得是否合理将直接影响到软件系统最终是否能够成功。毕竟，软件架构中包含了关于如何构建软件的一些最重要的设计决策，如系统分为哪些部分，各部分之间如何交互，等等；而采用这些设计决策，能否使最终开发出来的软件系统满足我们预期的运行期质量属性（如性能、可伸缩性、持续可用性、鲁棒性和安全性等）和开发期质量属性（如可扩展性、可重用性和可理解性）的要求，都是悬而未决的重大风险。

因此，在以架构为中心展开大规模的系统开发之前，切莫忘记先“问问电脑”——也就是通过将软件架构设计方案尽快实现为一个小的原型，并通过对该原型的测试和评审，来评估软件架构是否合理。

## 7.3 制定软件架构设计策略

策略对实践提供总体上的指导。对于有难度的工程（如软件工程）或有竞争性的目标（软件项目中时间、质量、范围和成本之间存在竞争）而言，策略往往是制胜的关键。

以表代文，表 7-1 概括了本章要讲述的 4 个软件架构设计策略。

表 7-1 软件架构实践中的问题及解决策略

关键点	问题	危害	策略	策略要点
是否遗漏了至关重要的非功能需求	对需求的理解不系统、不全面，对非功能需求不够重视	造成返工，项目失败	全面认识需求	弥补非功能需求的缺失
能否驯服数量巨大且频繁变化的需求	对于时间和质量的矛盾，办法不足，处理草率	耗时不少，质量不高	关键需求决定架构	“需求入架构出”的理解过于简单粗糙，不能适应实践要求
能否从容地设计软件架构的不同方面	架构设计方案覆盖范围严重不足，许多关键决定被延迟由实现人员仓促决定	开发混乱，质量不高	多视图探寻架构	架构是开展系统化团队开发的基础，应当为不同涉众提供指导和限制
是否及早验证架构方案并作出了调整	假设架构方案是可行的，直到后期才发现问题，造成大规模返工	造成返工，项目失败	尽早验证架构	架构设计方案应解决重大技术风险，并尽早验证架构



### 7.3.1 策略一：全面认识需求

软件架构强调的是整体，而整体性的设计决策必须基于对需求的全面认识；软件架构应该是稳定的，而遗漏了重要需求的架构设计面临的是返工的命运。

一言以蔽之，全面认识需求，是生产出高质量软件所必须的“第一项修炼”。

作为一个软件架构师，也不应对所有需求“胡子眉毛一把抓”，而是应全面认识需求——分门别类地将需求梳理清楚。

图 7-1 所展示的“需求空间分割图”揭示了全面认识需求的要求。要全面认识需求，意味着我们必须从不同级别来考察需求：组织级、用户级、开发级，还要对每个级别考虑不同类型的需求：功能需求、质量属性、约束。

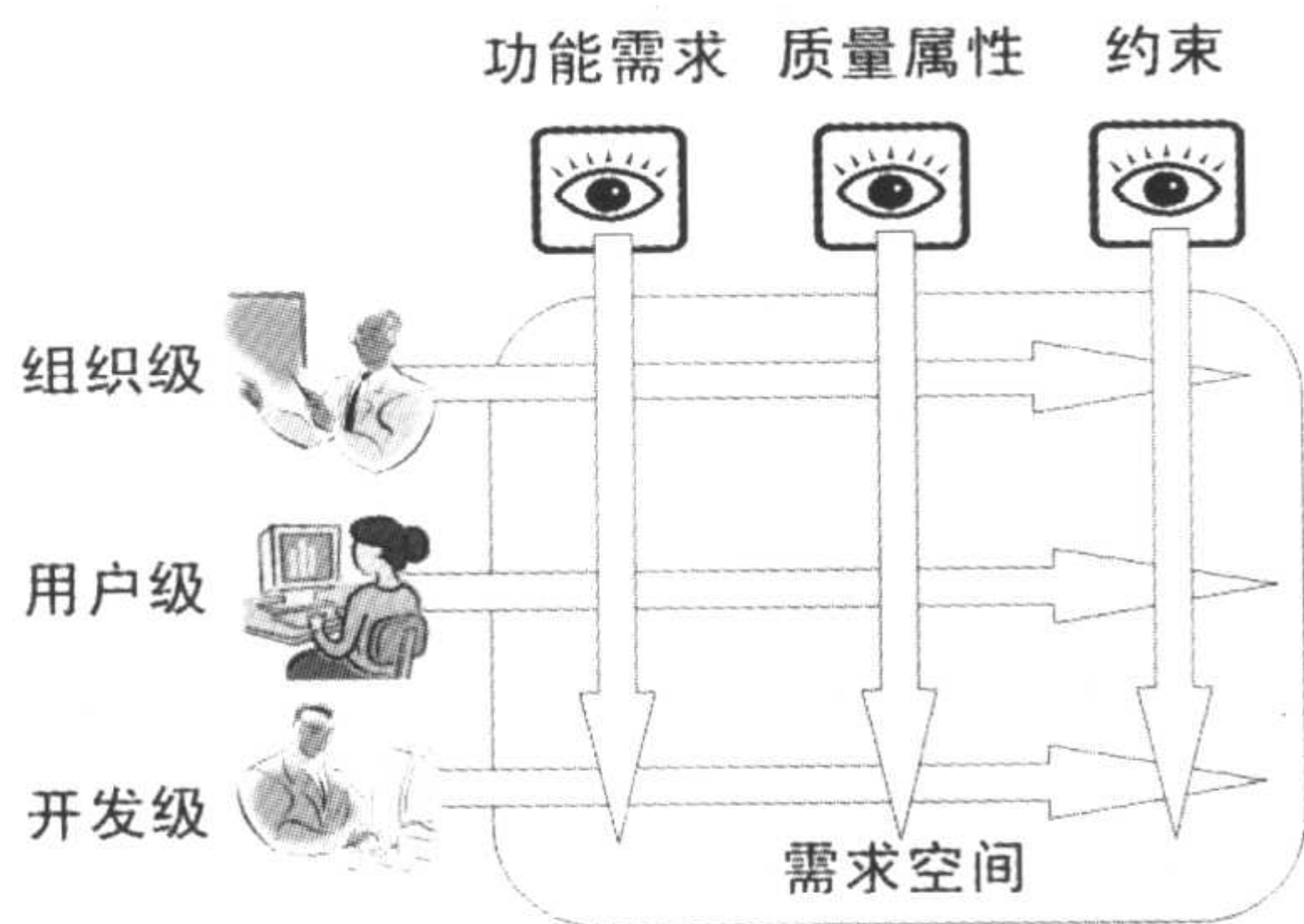


图 7-1 软件需求空间分割图

一方面，需求是分层次的。一个成功的软件系统，对客户高层而言能够帮助他们达到业务目标，这些目标就是客户高层眼中的需求；对实际使用系统的最终用户而言，系统提供的能力能够辅助他们完成日常工作，这些能力就是最终用户眼中的需求；对开发者而言，有着更多用户没有觉察到的“需求”要实现……

关注需求层次的实践意义在于，在需求之间建立起“可跟踪性”，避免因遗漏需求而造成软件“达不到要求”，也避免开发人员一厢情愿地为用户“制造”没有实际意义的无用功能。理解了需求分层的道理，软件人员在听到客户方的老板说“需求就是我希望这套软件能帮我赚更多的钱”时，就不会觉得好笑了，因为他知道这可能就是创建这套软件系统的商业目标，并会对其他需求和设计产生影响。

另一方面，需求应该被分为不同的类型。例如，一个网上书店系统的功能需求可能包括“浏览书目”、“下订单”、“跟踪订单状态”和“为书籍打分”等，质量属性需求包括“互操作性”和“安全性”等，而“必须运行于 Linux 平台之上”属于约束性需求之列。实践一再表明，忽视质量属性和约束性需求，常常导致架构设计最终失败。



总之，通过需求分类，将有助于全面认识需求、分门别类地把握需求和设计出高质量的软件架构。

全面认识需求还有一层含义，那就是应当在深思熟虑之后作出合适的需求权衡和取舍。一方面，众多质量属性需求之间往往会有冲突，我们必须权衡。另一方面，如果通过复杂设计所支持的变化根本不会发生，那么这种过度设计（Overengineering）就造成了资源的浪费并增加了开发的难度。有人主张不要预测未来，本书并不同意，本书认为应当有依据地支持未来变化，对变化的判断应该来自对需求及需求背景的深刻把握。

### 7.3.2 策略二：关键需求决定架构

软件架构师没有时间对所有需求进行深入分析，这是现实。软件架构师没有必要对所有需求进行深入分析，这是策略。

软件架构师需要做的就是：接受现实，采取策略。

具体而言，就是让关键需求决定架构。关键需求决定架构有两方面的内涵：

- 功能需求数量众多，应该控制架构设计时需要详细分析的用例的个数；
- 另一方面，不同质量属性之间往往具有相互制约性，于是我们自然应该权衡哪一部分质量属性是架构设计的重点目标。

等到所有需求都详细确定下来再开始架构设计是不现实的。需求来自于实践需要，而实践是发展的，所以“确定的需求”只是相对的。而且，项目何时交付往往是由客户业务的需要决定的，或者是市场形势决定的。因此在实际当中，我们一般在项目的业务目标及核心需求达成共识之后就开始架构设计；在这种情况下，关键需求决定架构的策略非常适合。

关键需求决定架构的策略有利于集中精力深入分析最为重要的需求。人的思维能力所能应对的复杂性是有限的，因此人们总是有意识地将问题分解、化简和转换。当我们把全部精力扑在相对少的需求上时，可以更为深入地分析这些需求，有利于得到透彻的认识，从而设计出合理的架构。

### 7.3.3 策略三：多视图探寻架构

从历史上看，对软件进行架构级的设计和软件复杂性增长有关，并成为战胜复杂性的主要手段之一。

分而治之的办法之所以有效，也在于它把复杂的问题划分成多个相对简单的子问题来分别解决。每个子问题仅强调整个问题的一部分或一个层面，对此，心理学研究早已表明，“强调能使经验活跃起来”，从而更有效地解决问题。

基于多视图的架构设计方法之所以有效，其原因正在于分而治之。它使软件架构师不必“在

同一时刻”处理逻辑层（Layer）、物理层（Tier）、子系统、模块、接口、进程、消息和协议等一股脑的问题，而是可以一次只围绕少数概念和技术展开，分别把精力放在软件的逻辑架构、物理架构、运行架构、开发架构和数据架构等不同的方面。

### 7.3.4 策略四：尽早验证架构

应对软件架构设计方案进行验证，而不仅仅是评审。

具体而言：

- 应真正地通过编码将软件架构实现；
- 应实际对架构原型进行测试，测试的重点是运行时质量属性，如性能、可伸缩性和鲁棒性等；
- 要认真评估架构原型的实现过程，以对软件架构的开发期质量属性给出评价，例如可理解性、可重用性和易修改性等；
- 传统意义上的架构设计评审依然需要，比如通过走查确保所有功能需求能够被支持；
- 要对架构设计的不足之处及时进行调整，并再次进行验证……

根据具体项目或产品的情况不同，有两种验证技术可以采用。

第一种，采用原型技术。确切地说，是通过开发一个垂直演进原型，来实现软件架构。原型技术有两种相互独立的分类方式。根据分析开发原型的目的是模拟系统运行时的概貌，还是纵深地验证一个具体的技术问题，可以将原型法分为两类：水平原型和垂直原型；根据所开发的原型是否被抛弃，又可以将原型法分为两类：抛弃原型和演进原型。第 17 章将详细讨论此话题。

第二种，采用框架技术。确切地说，是框架 + 垂直抛弃原型。这样做的好处是利于软件产品后期的演进和升级，对致力于开发某个领域的系列软件产品的企业尤其有帮助。

必须进行实际的测试，这一点很重要。否则，软件架构是否能达到运行期质量属性的要求就会不得而知，埋下了众多技术风险。

## 7.4 总结与强调

成功的架构设计是相似的，失败的架构设计各有各的原因。本章探究成功架构设计的“相似”之处——成功架构设计的关键要素。

架构设计不能遗漏至关重要的非功能需求，否则到最后不是项目失败，就是必须返工，因此软件架构师必须全面认识需求。

架构设计必须驯服数量巨大且频繁变化的需求，否则会非常被动，耗费大量精力，而软件架构质量却不高，因此软件架构师必须学会通过关键需求来主导架构设计。

架构设计涉及不同方面的设计决策，软件架构师应当采用基于多视图的架构设计方法。

架构设计的成果应及早验证，如果盲目假设架构方案是可行的，直到后期才发现问题，就会造成大规模返工乃至项目失败，因此软件架构师应注意尽早验证架构。





## 第 8 章 软件架构要设计到什么程度

---

众鸟高飞尽，孤云独去闲。

——李白，《独坐敬亭山》

架构被用作销售手段，而不是技术蓝图，这屡见不鲜（Too often, architectures are used as sales tools rather than technical blueprints.）。

——Thomas J Mowbray, 《What is Architecture》

我不会认为 Coding 和 Designing 是对立的。……而问题在于，那些设计人员的设计又往往是高来高去的扯淡，脱离实际情况，二者的矛盾就必然存在。

——猛禽，《设计不是一件玄事》

说得好！“设计不是一件玄事”。

但设计怎么就被搞成“玄事”了呢？举目四望，我们发现架构设计方案往往是高来高去，造成不同的人有不同的理解；而且，实际的开发工作还是得不到足够的指导，高技术风险依然大量存在；更有甚者，投标演讲归来，直接从演示的 ppt 上拷贝些东西，就想将之作为“架构设计方案”的全部，这样的“高效”要不得。

把设计搞得玄而又玄的结果是，很多影响全局的设计决策本应由架构设计来完成，却统统“漏”到了后边，最终到了大规模并行开发阶段才发现。这样一来，造成了“程序员碰头儿临时决定”的情况大量出现，软件质量必然下降，甚至还会导致项目失败。

于是我们不禁要问：软件架构到底要设计到什么程度？

本章希望通过如下努力回答上述问题：

- 首先，对软件架构的设计程度问题展开探讨，得出基本结论。从对“分而治之”的讨论入手，说明软件架构是团队开发的基础，从而，软件架构必须设计到“能为开发人员提供足够的指导和限制”的程度；
- 之后，从问题入手，认识高来高去式架构设计的“症状”。主要分析实际工作中常见

的架构设计程度不足的几种表现，找到要解决的问题；

- 然后，说明如何解决架构设计高来高去、指导不足的问题；
- 最后，结合案例，说明如何一步步地将架构设计落到实处。

## 8.1 软件架构要设计到什么程度

“分而治之”对于我们这个圈子而言是个相当老的话题了，但我们见到的大多数对“分而治之”的讨论都只是泛泛而谈。本书想再就这个老话题说上几句，并以此为基础阐明软件架构设计在整个软件开发周期中的重要地位，从而说明软件架构必须设计到“能为开发人员提供足够的指导和限制”的程度。

### 8.1.1 分而治之的两种方式

为什么要分而治之？简单说，就是问题太复杂了，超出了人们能够“一蹴而就”的范围。

《人月神话》的作者 Frederick P. Brooks 对软件复杂度的论述相当深刻，“软件的复杂度是根本属性，不是次要因素”的论断影响深远。软件复杂度一直以来都是软件开发方法着力解决的问题之一，不止一个大师级人物曾有类似表示：采用面向对象方法的“最重要的原因”是它可以帮助我们解决更复杂的问题（而不是更好的可重用性）。

面对一个复杂的问题，我们当如何进行分而治之呢？策略有二，如图 8-1 所示。

一是先不把问题研究得那么深，那么细，浅尝辄止，见好就收。这种分而治之的方式称为“按问题深度分而治之”。

二是先不研究整个问题，而是研究问题的一部分，分割问题，各个击破。这种分而治之的方式称为“按问题广度分而治之”。

例如，接口和实现分离，就是“按问题深度分而治之”一个很好的例子。如图 8-2 所示，在架构设计之时，我们往往无需深入到一个子系统的实现细节中去，而是分而治之，先确定该子系统的接口。接口的设计在整个架构设计中占有重要地位，它定义了一个子系统为其他子系统所提供服务的契约。软件架构通过明确每个子系统所要实现的接口及所要调用的接口，为我们展现了一个软件系统如何分割为多个相互协作的子系统。



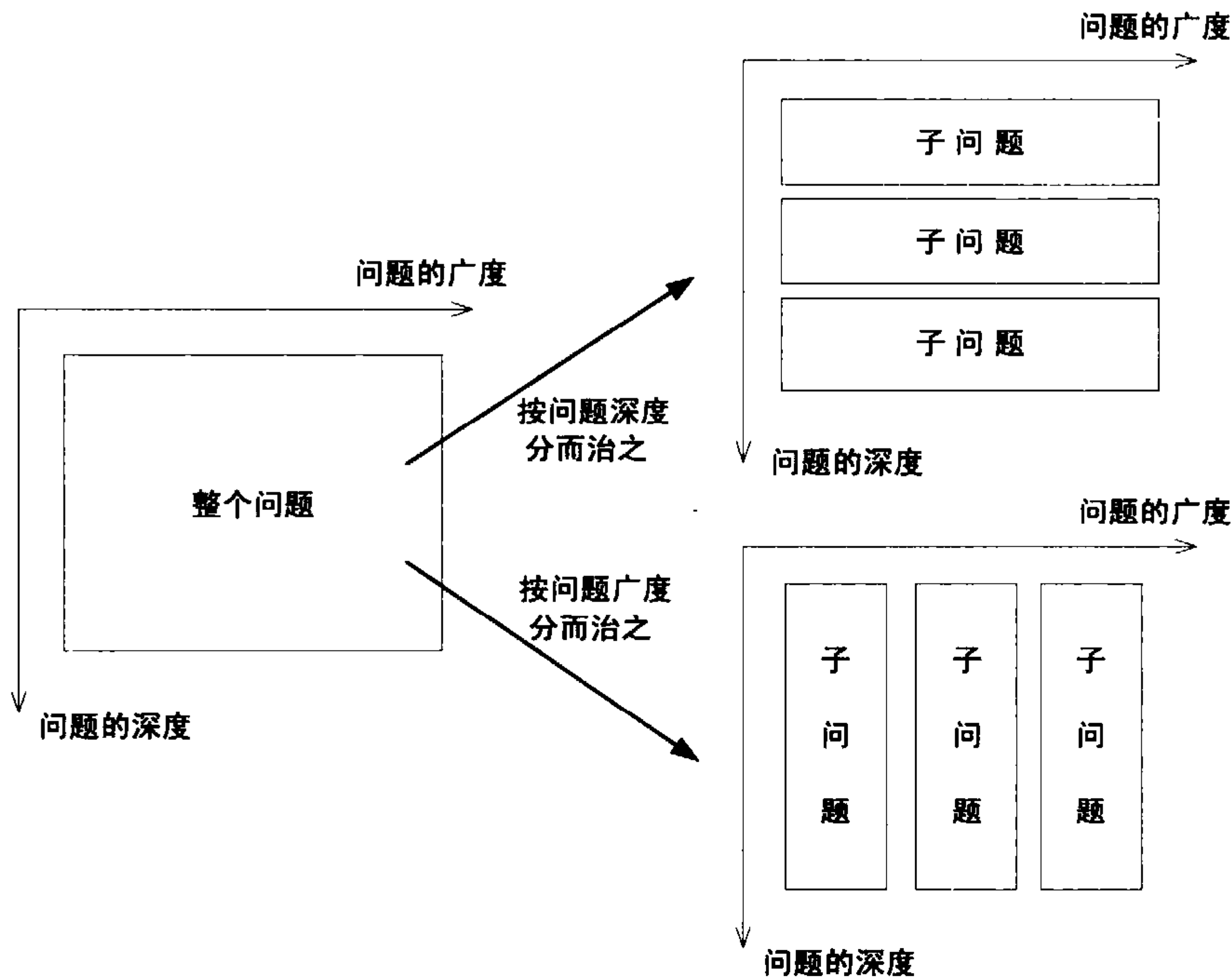


图 8-1 分而治之的两种方式

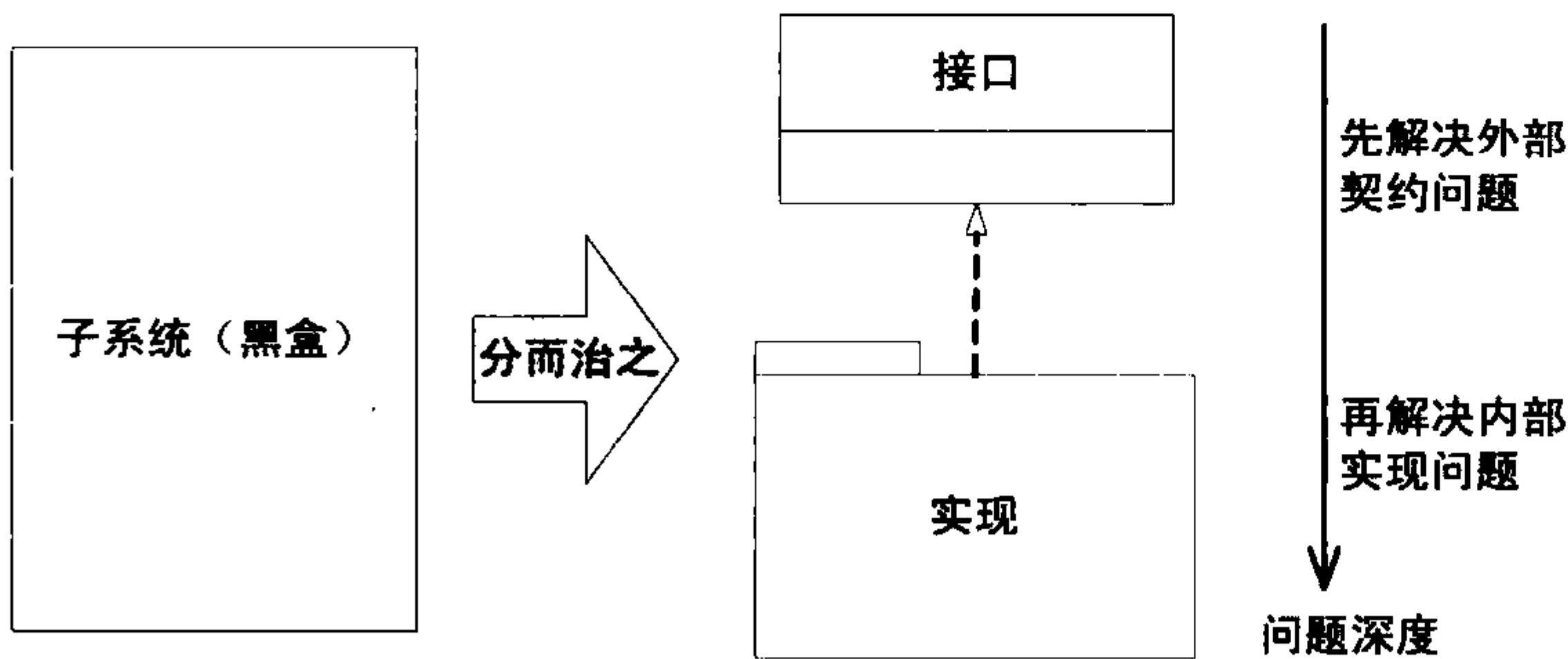


图 8-2 分而治之的例子：接口和实现分离

“按问题广度分而治之”的例子也很常见，比如展现层、业务层和数据层的开发往往需要不同的技术，可以分派给不同的小组承担等，不一而足。

8.1.2 架构设计与详细设计

随着软件设计工作越来越复杂，将架构设计和详细设计分离已成为普遍的做法。Garlan 就曾指出：“随着软件的规模和复杂度增加，算法和数据结构以外的设计问题就会出现：设计和制定系统整体结构将成为新的一类问题……这是软件架构层次的设计。”

将设计分为架构级设计和详细设计，是对“按问题深度分而治之”思想的运用：

- 所谓架构设计，就是关于如何构建软件的一些最重要的设计决策，这些决策往往是围绕将系统分为哪些部分，各部分之间如何交互展开的；
- 而详细设计针对每个部分的内部进行设计。

可以这么说，软件架构设计应当解决的是全局性的、涉及不同“局部”之间交互的设计问题，而不同“局部”的设计由后续的详细设计负责。于是，在软件架构所提供的“合作契约”的指导下，众多局部问题被很好地“按问题广度分而治之”了——对局部的详细设计（及编码实现）完全可以并行进行。

总之，这种先确定软件架构，而后基于软件架构进行并行开发的做法，综合利用了两种分而治之的方法（如图 8-3 所示），利于控制复杂性，提高开发效率。这是一种值得推崇的开发方式，常被称为“以架构为中心的开发方法”。

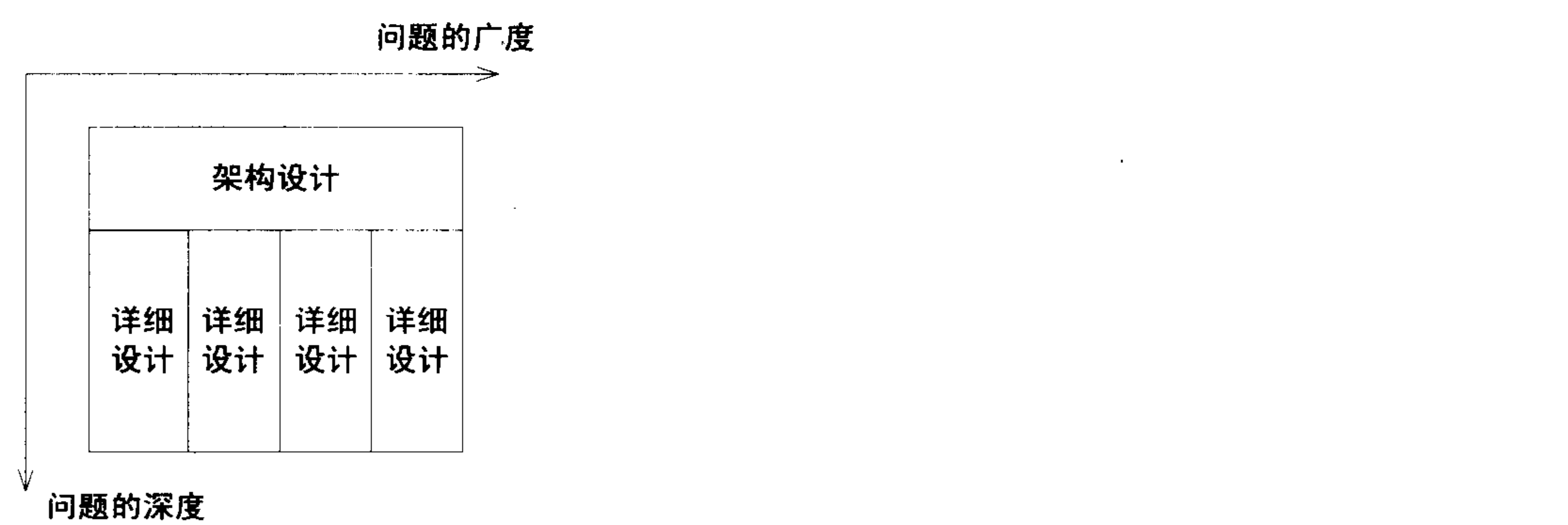


图 8-3 架构设计和详细设计的关系

8.1.3 软件架构是团队开发的基础

正如大家所看到的，因为软件变得越来越复杂了，所以单兵作战不再是普遍的软件开发方式，取而代之的是团队开发；而团队开发又反过来使软件开发更加复杂，因为现在不仅仅有技术复杂性的问题，还有管理复杂性的问题。

面对“技术复杂性”和“管理复杂性”这样的双重困难，以架构为中心的开发方法是有效的途径：

- 一方面，软件架构从大局着手，就技术方面的重大问题作出决策，构造一个具有一定抽象层次的解决方案，而不是将所有细节统统展开，从而有效地控制了“技术复杂性”。可以看出，软件架构设计这一步没有“把问题研究得那么深、那么细”，属于“按问题深度分而治之”。对此，Ivar Jacobson 给我们提供了非常形象的说法，“软件系统的架构涵盖了整个系统，尽管架构的有些部分可能只有‘一寸深’”（出自《统一软件开发过程之路》）；
- 另一方面，有了软件架构设计方案之后呢？因为“架构中包含了关于各元素应如何彼此相关的信息”（Len Bass 语），从而可以把不同模块分配给不同小组分头开发，而软件架构设计方案在这些小组中间扮演“桥梁”和“合作契约”的作用。每个小组的工作覆盖了“整个问题的一部分”，各小组之间可以互相独立地进行并行工作，这种“分割问题，各个击破”的策略，属于“按问题广度分而治之”。

这两方面是相辅相成的关系。具体而言，正因为软件架构是大规模开发的基础，所以架构中应包含软件系统的各元素如何彼此相关的设计决策；也正是因为软件架构中包含了软件系统如何组织等关键决策，才使得它能够成为大规模开发的基础。

这样一来，模块的技术细节被局部化到了小组内部，内部的细节不会成为小组间协作沟通的主要内容，理顺了沟通的层次。另外，对“人尽其才”也有好处，不同小组的成员需要精通的技术各不相同。例如，用户界面层的开发小组需要了解将使用的用户界面工具包；数据管理层的开发小组需要熟悉相关的数据库、持久工具或者使用的文件系统；系统交互层的开发小组需要了解通讯协议和用到的中间件产品等。

由此可见，软件架构为开展系统化的团队开发奠定了基础（如图 8-4 所示），为解决“管理复杂性”提供了有力的支持。对此，Barry Boehm 曾经明确指出，“如果一个项目的系统架构（包括理论基础）尚未确定，就不应该进行此系统的全面开发。”

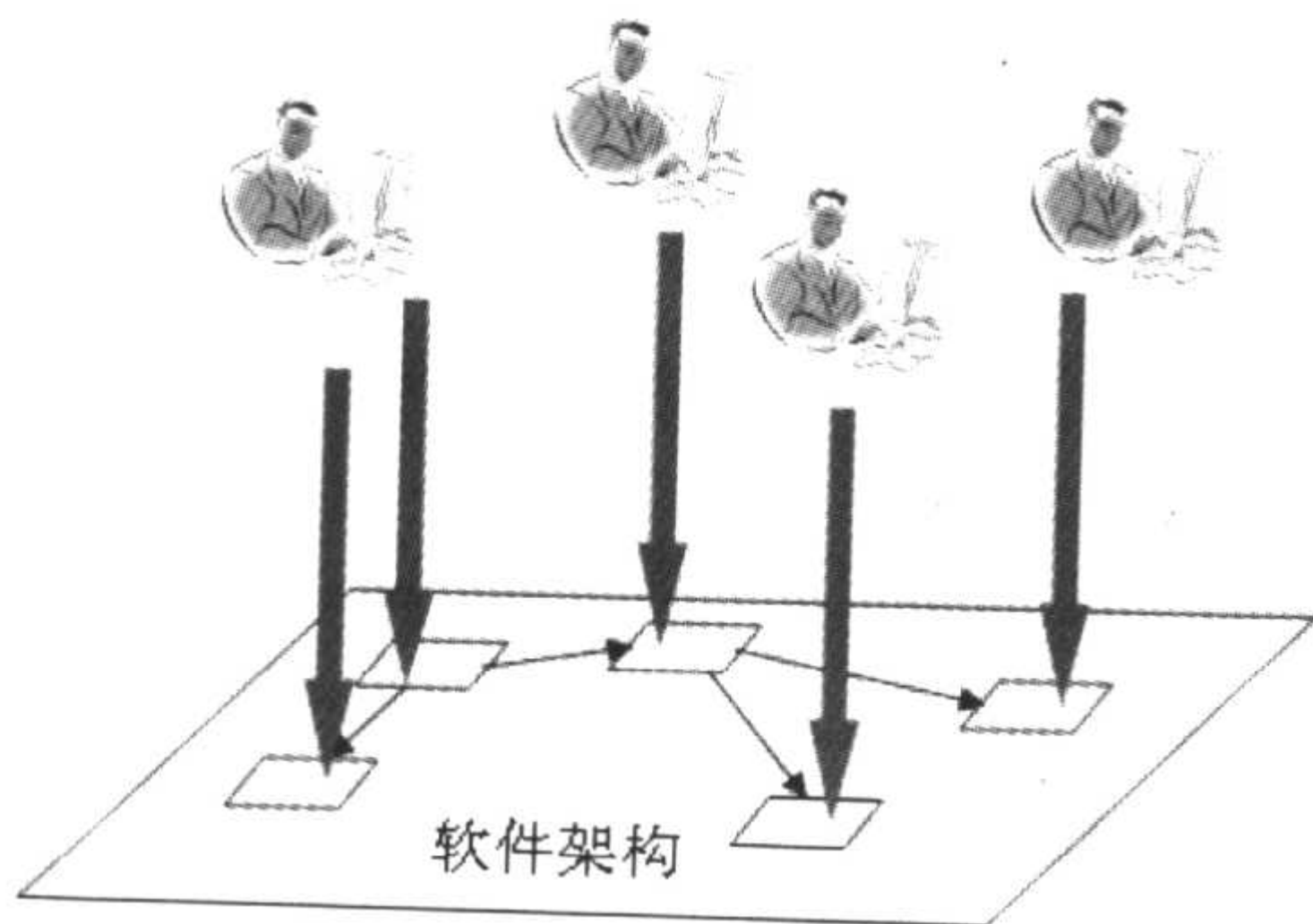


图 8-4 软件架构为团队开发奠定了基础



### 8.1.4 架构设计要进行到什么程度

既然软件架构是团队开发的基础，那么它就应该比较明确地规定后期分头开发所必须的公共性的设计约定，从而为分头开发提供足够的指导和限制。

另一方面，具体的架构设计程度还会因软件项目的不同而不同，例如，航空航天领域中的软件系统对系统可靠性要求非常高，这种情况下对架构设计详细程度的要求也会比较高；同时，架构设计的程度也会受团队具体情况的影响，有类似项目经验的团队可以适当放宽标准，而分布团队或涉及外包的情况则应更强调架构的明确性。图 8-5 对上述说明进行了总结。

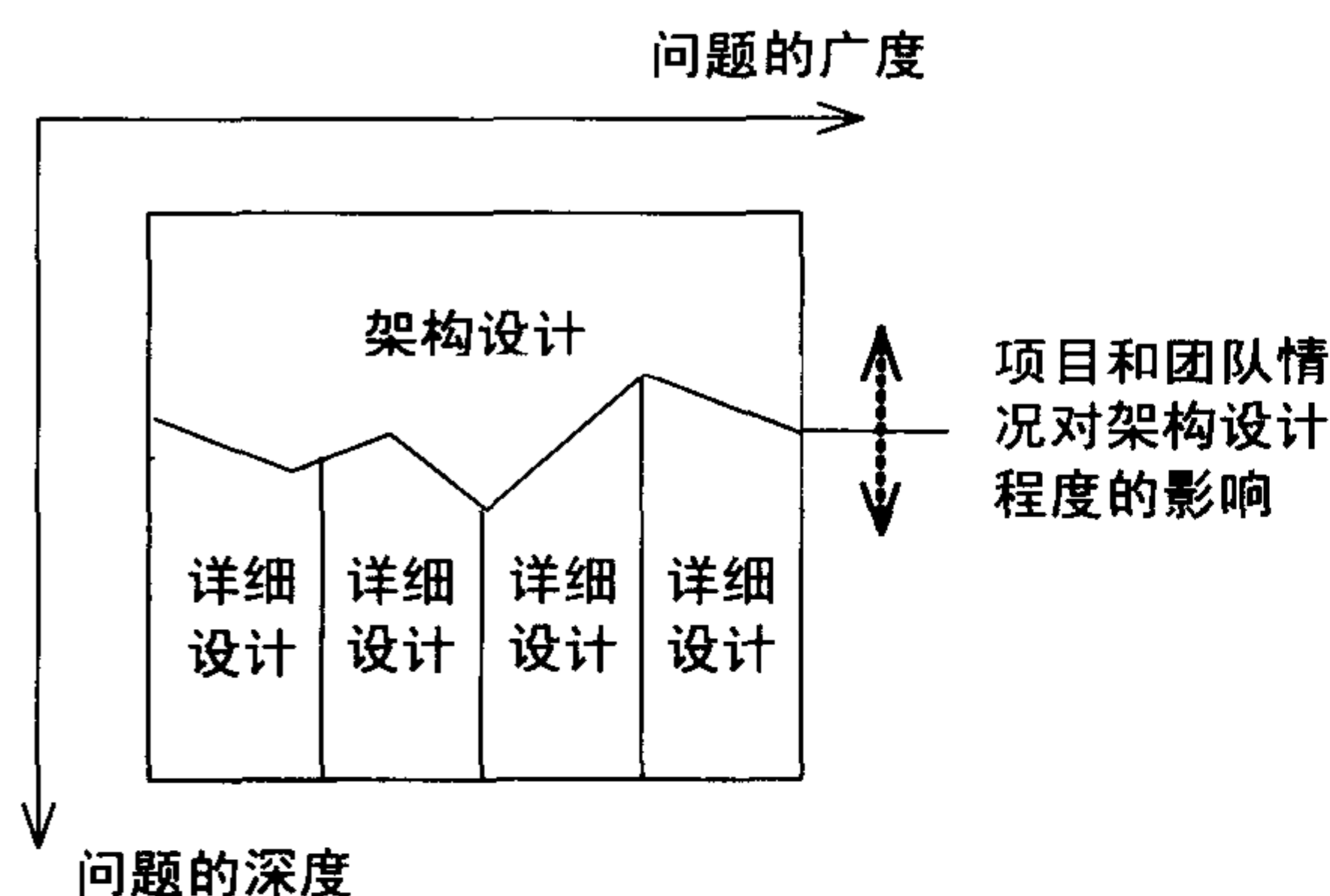


图 8-5 架构设计要进行到什么程度

从图中也可以看出，架构设计对软件的不同部分的设计程度并不是整齐划一的。例如，通信机制、持久化机制和消息机制等对应的公共模块，在架构设计时会设计得比较深入，因为它们较多地涉及软件的不同部分之间的交互；而具体的业务功能模块在架构设计中往往设计程度不深。

总之，关于软件架构到底要设计到什么程度，可以归纳为两句话：

- 由于项目的不同、开发团队情况的不同，软件架构的设计程度会有不同；
- 软件架构应当为开发人员提供足够的指导和限制。

## 8.2 高来高去式架构设计的症状

所谓“高来高去式架构设计”，是指不能为开发人员提供足够的指导和限制的那种架构设计方案。高来高去式架构设计现象极为普遍，它可能带来的危害包括：

- 缺少来自架构的足够的指导和限制，开发人员在进入分头开发阶段之后会碰到很多问

题，并且容易造成管理混乱，沟通和协作效率低；

- 对软件质量非常关键的全局性设计决定被拖延到分头开发期间草率决定，严重降低了软件产品质量；
- 没有完成化解重大技术风险的职责，可能造成整个项目失败；
- 团队成员对架构师意见很大，团队士气受到打击。

“发现问题是解决问题的一半”，下面为高来高去问题归类。根据笔者的观察，高来高去式的架构设计大致有如下三种表现。

**症状一：缺失重要架构视图。**为了给团队的不同角色提供切实的指导，软件架构师应从不同视图进行架构的设计和归档。如果忽视了在某重要架构视图方面的设计，就会遗漏重要设计决策，具体的开发工作将会陷入缺乏应有指导和限制的困境。

**症状二：浅尝辄止、不够深入。**架构设计方案过于笼统，基本还停留在概念性架构的层面，没有提供明确的技术蓝图。于是，全局性的设计决策最后由具体开发人员从局部视角考虑并确定下来，造成技术和管理上的种种问题。

**症状三：名不副实的分层架构。**通过分层将软件系统模块化之后，就迫不及待地喊出“分层架构”的口号，对各层之间交互接口和交互机制的设计严重不足。这种情况屡见不鲜。

### 8.2.1 缺失重要架构视图

如你所知，由于角色和分工不同，整个软件团队以及客户等涉众各自需要掌握的技术或技能存在很大差异。这就使得他们为了完成各自的工作，需要了解整套软件架构决策的不同子集。基于多重视图开展软件架构设计的方法，就能够解决上述问题。

实际经验表明，越是复杂的系统，越是需要从多个方面进行架构设计，这样才能把问题研究和表达清楚。Grady Booch 在其著作《UML 用户指南》中指出：“如果选择视图的工作没有做好，或者以牺牲其他视图为代价只注重一个视图，就会冒掩盖问题以及延误解决问题（这里的问题是指那些最终会导致失败的问题）的风险。”

为不同的软件系统进行架构设计时，对不同的架构视图的重视程度并不相同。例如，用于反映软件系统运行时的动态结构，以及组成软件系统的目标程序如何部署到硬件上的物理架构视图，对于分布式系统的架构设计来说是必不可少的。再例如，现在大量采用现成框架（比如众多开源框架）进行开发的软件系统越来越多了，这时开发架构视图就显得极为必要——开发架构关注程序包，不仅包括要编写的源程序，还包括可以直接使用的第三方 SDK 和现成框架和类库，以及开发的系统将运行于其上的系统软件或中间件。对此，在第 5 章“架构设计的 5 视图法”中已有详细讨论。

“缺失重要架构视图”的一种表现是，认为软件架构设计完全是用例驱动的，片面强调用例描述的功能需求。此时，架构设计对非功能需求关注不够，既没有深入设计软件的运行架构（对

性能、可伸缩性、持续可用性等运行期质量属性很重要), 也没有深入设计软件的开发架构(对可扩展性、可重用性、可测试性和可移植性等开发期质量属性很重要)。软件开发人员会感觉到架构设计方案离他们很“遥远”, 既没有说明程序包的组织结构, 也没有明确架构设计中的关键抽象和所采用框架(Framework)的关系, 等等。

## 8.2.2 浅尝辄止、不够深入

此时, 架构设计方案往往过于笼统, 基本还停留在概念性架构的层面, 没有提供明确的技术蓝图。

概念性架构虽说有用, 但“有用”不代表“够用”。概念性架构对开发人员的指导和限制是不够的。为了化解高技术风险, 证明技术上的可行性, 必须充分考虑技术, 例如关键机制、核心技术和第三方框架等都应明确。

另外, 投标或售前演示级别的架构方案对市场活动或许足够了, 但对实际的开发却显得过于笼统。比如, 对如何达到高安全性目标, 仅提出了采用传输加密、存储加密和数字签名等技术, 但并未明确这些技术如何影响软件系统的其他部分, 它们是否通过统一的服务接口封装, 别的模块是直接调用它们还是利用 AOP 机制松耦合地触发等问题。同样, 为了达到高可扩展性的要求, 轻描淡写地“采用模块化设计”对实际开发也过于笼统。

无论上述的哪一种情况, 它们造成的危害都是相同的: 架构设计阶段遗漏了全局性的设计决策, 到了大规模开发实现阶段, 这些设计决策往往被具体开发人员从局部视角考虑并确定下来。如此一来, 就会在模块协作方面出现问题, 而且公共的服务模块也未能被识别出来。例如, 如图 8-6 所示, 左图仅识别出了多个模块, 由于没有深入考察模块之间的协作机制问题, 忽视了“拓扑显示”和“告警通知”都应通过“消息机制”即时接收“网络设备模型”的变化这一重要情况, 从而没有将公用的“消息机制”模块单独抽取出来, 造成开发中的诸多问题。

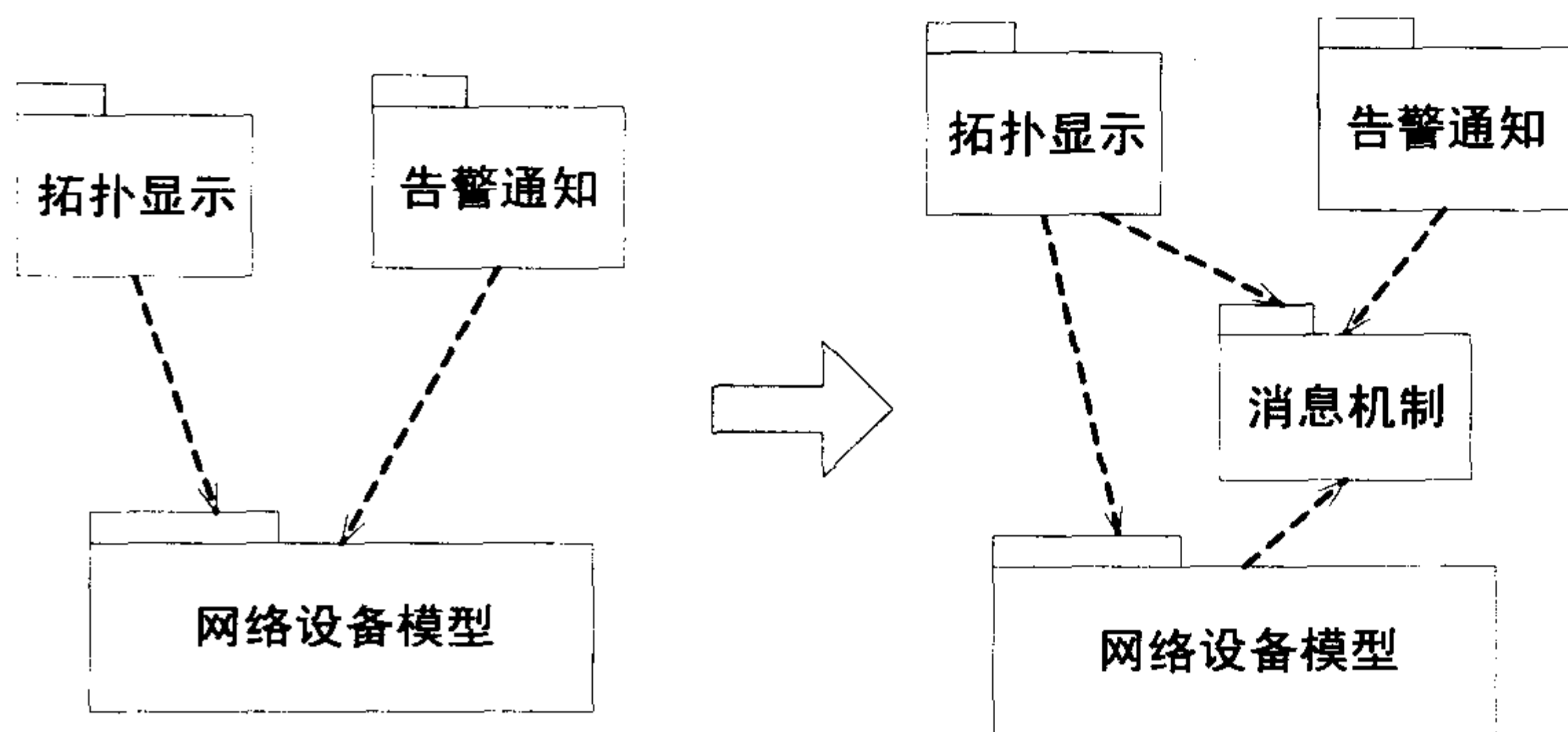


图 8-6 不够深入就不能发现公共模块



### 8.2.3 名不副实的分层架构

“名不副实的分层架构”可以认为是“浅尝辄止、不够深入”式架构设计的一种具体表现，但由于分层架构的应用太广泛了，几乎无处不在，因此我们把它单独列出并着重讨论。

“名不副实的分层架构”是指那些号称采用分层架构，却仅用分层来进行职责划分，而没有规划层次之间的交互接口和交互机制的情况。而经典的分层架构属于“调用—返回”式的架构模式，和“主程序—子程序”架构模式同属一种类型。可以说，缺失交互接口和交互机制的分层架构中，其实“层”已退化成笼统意义上的“职责模块”了。

很多道理是相通的。一个公司，它的组织结构图就挂在墙上——再清晰不过了，但公司的运营却很混乱，这是为什么呢？来了任务、出了事情不清楚找哪个经理，此谓接口不清；日常的汇报体系如何，突发事件如何处理也未明确，此谓机制不明。其实，这何尝不是一种架构不够明确的表现呢？

对软件系统也是同样，面对缺失交互接口和交互机制的分层架构，许多开发人员依然得不到足够明确的指导。

“名不副实的分层架构”常见于各大报纸、各种市场材料。由于它们常冠以“总体架构”等名称出现，而不是实际上的“软件架构之冰山一角”，所以，有些没有经验的架构师会误认为他们要设计的也是这种“高来高去式的软件架构”。

让我们来重温一下学生时代的“反证法”：

- (1) 假设，市场材料里的软件架构不是高来高去的，而是程度足够深入的；
- (2) 那么，竞争对手可以轻松地从市场材料里获知你的核心技术；
- (3) 但是，是没有人会把核心技术写到市场材料里公开宣扬的；
- (4) 因此，假设不成立，这意味着市场材料里的软件架构是高来高去的。

## 8.3 如何克服高来高去症

批评应有理有度。

可以这么说，高来高去式的架构本身并没有错，它们往往属于概念性架构的范畴；一来它对于市场宣传来讲已经足够了，二来它往往是循序渐进地进行软件架构设计的良好起点。但是，如果停留在高来高去的架构设计上止步不前，并以之作为最终的架构设计方案，就会为后期开发埋下重大风险。

以表代文，将高来高去式架构设计的症结、问题与对策归纳为一张表，如表 8-1 所示：

表 8-1 高来高去式架构设计的症结、问题与对策

症结	问题	对策
缺失重要架构视图	遗漏了对团队某些角色的指导	针对遗漏的架构视图进行设计
浅尝辄止、不够深入	将重大技术风险遗留到后续开发中	设计决策须细化到和技术相关的层面
名不副实的分层架构	只用了分层架构的分层概念来进行职责划分，没有明确层次之间的交互接口和交互机制	步步深入，明确各层之间的交互接口和交互机制

## 8.4 网络管理系统案例：如何将架构设计落到实处

本节通过网络管理系统架构的“设计片断”，演示了如何将架构落到实处的。

### 8.4.1 网管产品线的概念性架构

本案例的网络管理软件是作为软件产品线出现的，应充分考虑多个产品之间的重用，建立共享的产品线架构。

请回顾第 6 章的图 6-9 所示的“网管系统产品线的概念性架构图”……此概念性架构清晰地定义了架构的大局，并充分考虑了可移植性、可扩展性、性能和版权等商业因素，以及较长的生命周期等商业目标这些因素，制定了相应的高层设计决策。

但是，概念性架构毕竟是概念性架构，并不能为实际的开发工作提供足够的指导和限制。下面我们来进行实际架构层面的设计；篇幅有限，仅涉及识别功能块、规划功能块的接口、明确功能块之间的使用关系和使用机制等话题。

### 8.4.2 识别每一层中的功能模块

抽象的职责最终必须由具体的功能模块来承担。我们接下来必须识别每一层中的功能模块。作为例子，图 8-7 展示了逻辑层中又包含了哪些功能模块。

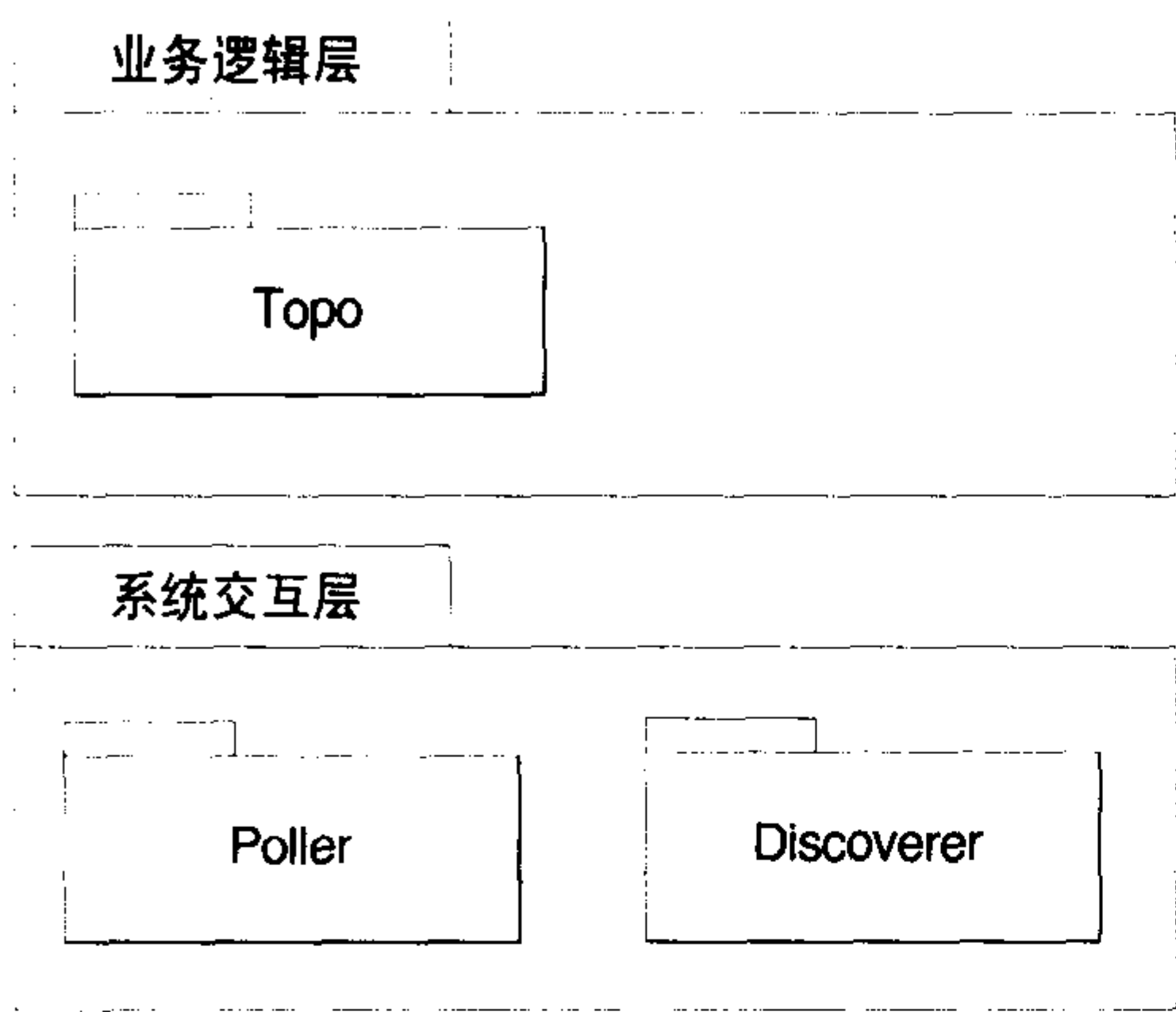


图 8-7 识别每一层中的功能模块

我们关注 3 个重点模块：网络拓扑模块、网络设备发现模块和网络设备状态轮询模块。这 3 个模块所担任的职责，我们通过图 8-8 所示的 CRC 卡来说明。

<div>网络拓扑模块</div> <div>职责<ul style="list-style-type: none"><li>●向设备发现模块发送发现命令</li><li>●接收设备发现模块的汇报</li><li>●接收状态轮询模块的汇报</li></ul></div>		<div>协作者<ul style="list-style-type: none"><li>●设备发现模块</li><li>●状态轮询模块</li></ul></div>	
<div>设备发现模块</div> <div>职责<ul style="list-style-type: none"><li>●接受网络拓扑模块的发现命令</li><li>●向网络拓扑模块发送所发现设备的信息</li></ul></div>		<div>协作者<ul style="list-style-type: none"><li>●网络拓扑模块</li></ul></div>	
<div>状态轮询模块</div> <div>职责<ul style="list-style-type: none"><li>●向网络拓扑模块发送设备状态信息</li></ul></div>		<div>协作者<ul style="list-style-type: none"><li>●网络拓扑模块</li></ul></div>	

图 8-8 功能模块的 CRC 卡

8.4.3 明确各层之间的交互接口

在分层架构中，下层对于上层应尽量做到“黑盒”封装。这样一来，为每一层规划接口就显



得非常重要。下面我们需要定义各层之间的交互接口，如图 8-9 所示。

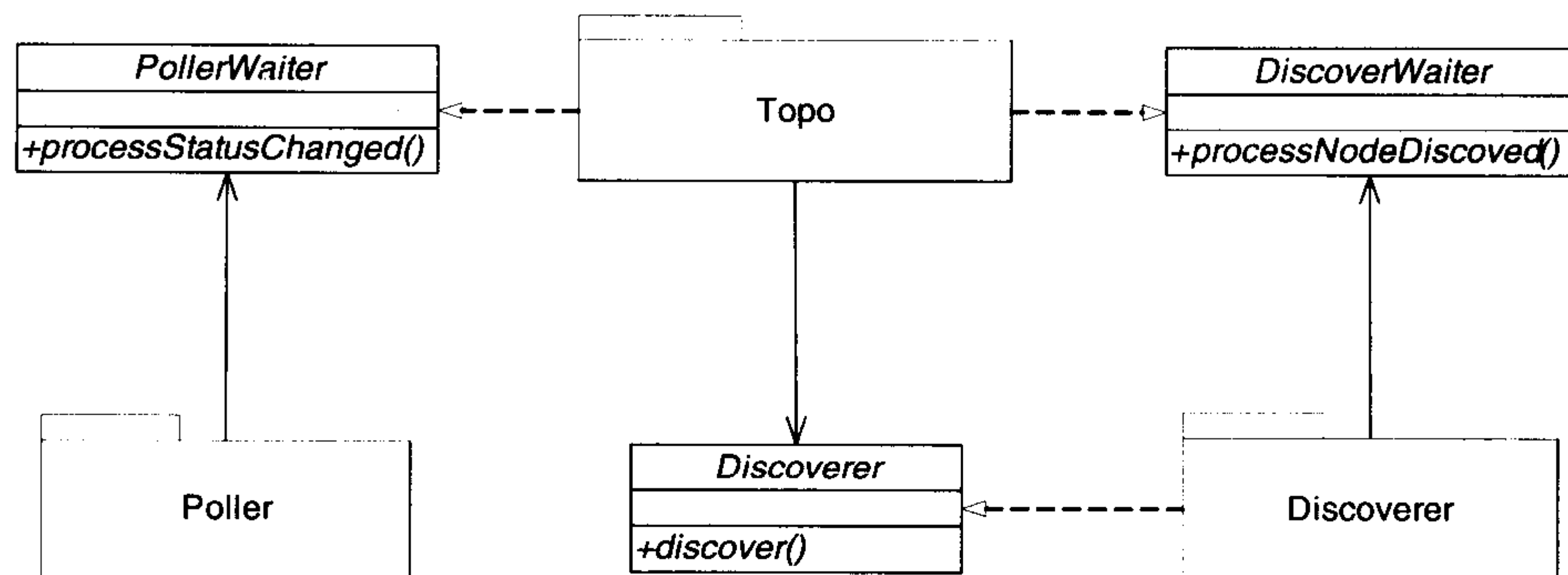


图 8-9 明确接口

#### 8.4.4 明确各层之间的交互机制

更进一步，有了接口，就可以明确交互机制了。如图 8-10 所示，设计中采用的其实是一种基于方法调用的事件机制。

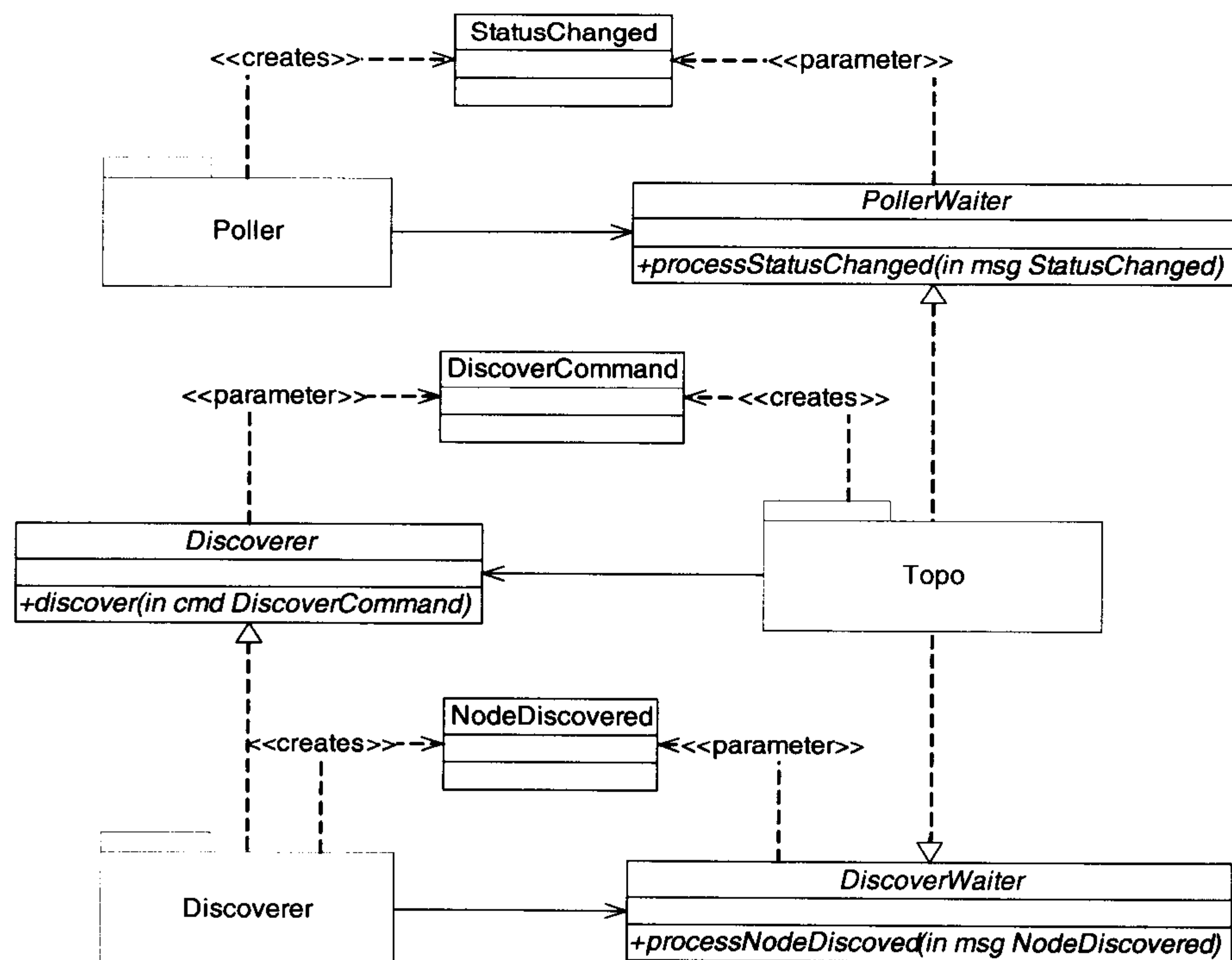


图 8-10 明确各层之间的交互机制

### 8.4.5 案例小结

我们没有深入描述设计时所考虑的细节，而是着重描述了设计从不明确步步走向明确的过程。

我们还必须说明，如何将架构设计落到实处，其实会根据架构设计视图的不同而不同。这是因为，不同的软件架构视图关注的对象不同（从模块—到程序包—到进程—到数据库表—到物理节点等），从而设计手段也就会有差异。而本案例本质上仅关注了一个架构视图：逻辑架构视图的设计。

## 8.5 总结与强调

---

本章讨论了一个对软件开发和管理都有深刻影响的问题：软件架构要设计到什么程度？如果说前面讨论的架构视图是关注了软件架构的横向宽度，那么本章就是关注软件架构的纵向深度。

关于软件架构到底要设计到什么程度，可以归纳为两句话：

- 由于项目的不同、开发团队情况的不同，软件架构的设计程度会有不同；
- 软件架构应当为开发人员提供足够的指导和限制。





## 第 9 章 软件架构设计过程

---

东临碣石，以观沧海。

——曹操，《观沧海》

作为职业软件人，我们都寻求使用一种有效而经济的过程来建造一个能够工作的、有用的产品。

——Grady Booch

好的过程清晰地定义任务。任务焦点放在结果上而不是细节上，项目可有效地进行，但仍然可以适应一些非常事件和环境的变化。

——Stephen R. Palmer，《特征驱动开发方法与实践》

从本章开始，我们将讨论软件架构设计过程。

我们将在通用的软件过程基础上，归纳软件架构师“自己的过程”，并把第 7 章讨论的架构设计策略结合进来。只有这样，软件架构设计过程所提供的指导才能有效，并更贴近实践。

### 9.1 打造有效的架构设计过程

---

#### 9.1.1 一般的软件过程

一般来讲，软件开发过程包括如下 5 个阶段：概念化阶段、分析阶段、架构设计阶段、并行开发与测试阶段、验收与交付阶段（如图 9-1 所示）。

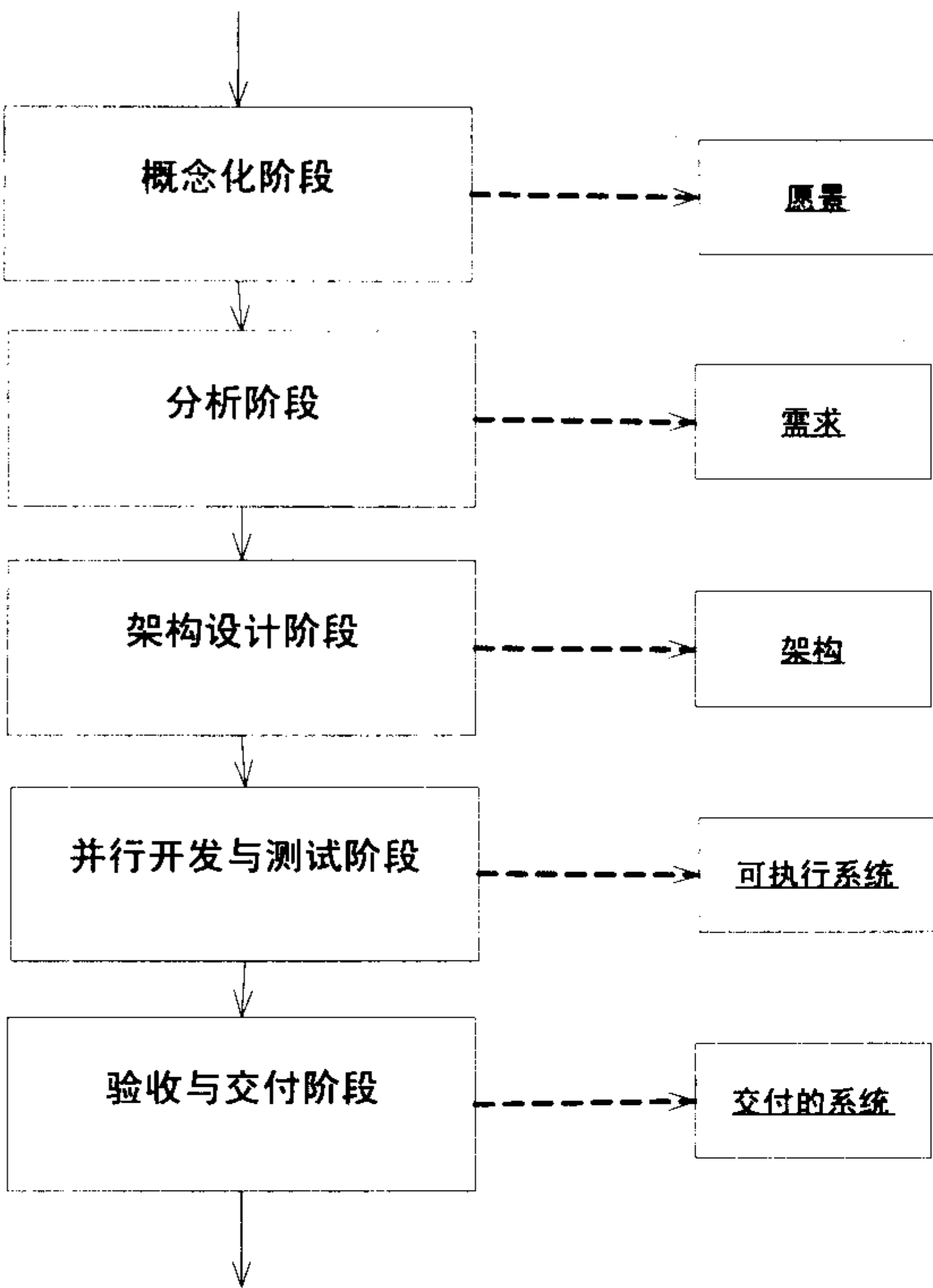


图 9-1 一般的软件过程

概念阶段要解决项目的起源问题，主要针对项目目标、主要特性、功能范围和成功要素等进行构思并达成一致。对规模较小的项目而言，概念化阶段所做的工作可以很少，例如通过《业务目标列表》的形式将项目希望达到的业务目标列出来即可。对规模较大的项目而言，概念化阶段就正规得多。但由于不同企业的开发管理方式之间存在较大差异，所产生的文档可能会有所不同：

- 《愿景与范围文档》（Vision and Scope Document）。该文档应说明项目的业务机遇、业务目标、业务风险以及为客户带来的价值，并通过列举项目主要特征和依赖环境的形式简要描述项目愿景的解决方案，还应该说明项目范围、局限性、成功与否的关键因素等诸多问题；
- 《市场需求文档》（Marketing Requirement Document，MRD）。这份文档和《愿景与范围文档》很相似，但更强调市场机遇、市场要求和竞争对手等因素。MRD 在一些以产品开发为主（而不是项目开发）的软件公司中较为常用；

- 《项目立项书》。项目型公司经常采用该文档。文档覆盖的内容包括项目背景、需求来源、高层需求、项目概况和可行性说明等。

分析阶段的目的是明确需求，并以《软件需求规格说明书》的形式记录下来。其中既包含软件系统必须完成的功能，又包含应满足哪些质量属性要求（如性能、安全性等），还须明确对设计实现有哪些约束性需求。

架构设计阶段要在较高的抽象层次上制定出解决方案，即设计软件架构。软件架构从系统如何规划、如何开发和如何运行等角度揭示了软件系统的结构和机制，为后面的具体开发工作提供足够的指导和限制。

并行开发与测试阶段动用的资源是最多的。此阶段中，我们以软件架构为基础，进行系统化的开发与测试。由于软件架构对组成软件系统的各个部分的职责和它们之间的交互已有清晰的定义，因此各个开发小组可以相对独立地工作。不同部分的工作成果应持续集成，增量发布，以便不断测试，不断评审，不断反馈和不断改进。最终，发布功能完整的可执行系统。

在验收与交付阶段，最终的产品将接受客户方或相关管理部门的验收。如果有需要，还要进行部署，上线，甚至已有数据的移行等工作。

上述软件过程之所以被称为“一般的软件过程”，是因为它是项目经理、架构师、开发人员、测试人员等所有人共同遵守的过程，是一个在许多方面都“大而化之”的“公共软件过程”。

### 9.1.2 架构师自己的架构设计过程

不难理解，一般的软件过程只重共性没有个性，它难以对承担某项工作的具体角色提供充分指导。例如，这个公共软件过程一般不会涉及测试计划、测试设计、测试执行和测试评估这样详细的步骤，而在测试人员看来这些步骤却是必不可少的。

同样地，从软件架构师的角度而言，他希望能有能够切实指导自己实际工作的架构设计过程。这倒不是说架构设计过程和整个软件开发过程有不一致的地方，而是说前者应当比后者有更多架构设计的“细节”。图 9-2 所示为软件架构设计过程的总览图，图中不仅显示了软件架构师在架构设计阶段应负责的活动，还显示了架构师在分析阶段应参与的相关活动。

可以看出，架构设计的开展非常依赖其上游活动。总体而言，这些上游活动包括需求分析和领域建模。

**需求分析。**毋庸置疑，在没有全面认识需求并权衡不同需求之间相互影响的情况下，设计出的架构很可能有问题。



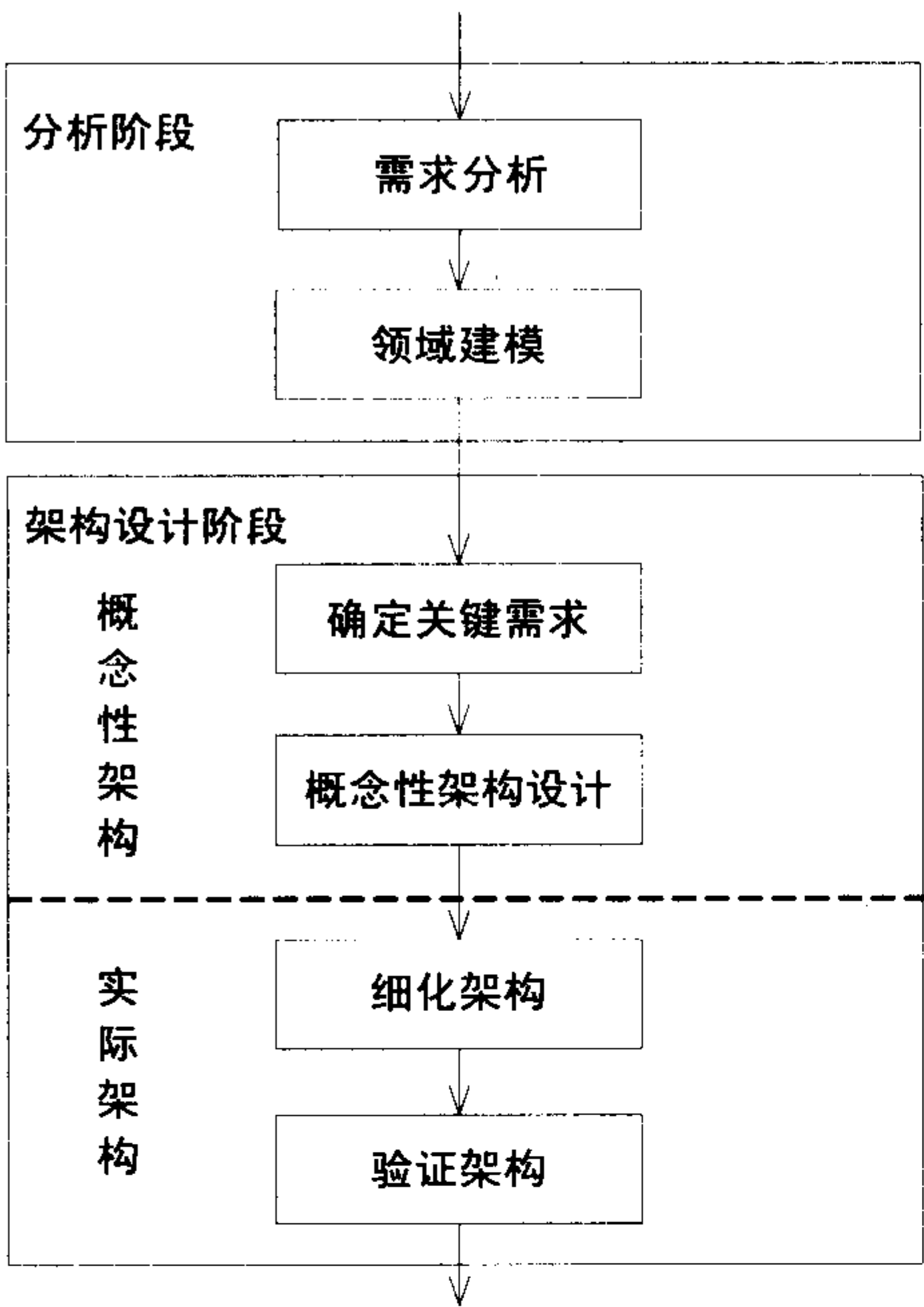


图 9-2 软件架构设计过程

**领域建模。**领域建模的目的是：透过问题领域的重重现象，捕捉其背后最为稳固的领域概念及这些概念之间的关系。在项目前期，所建立的领域模型将为所有团队成员之间、团队成员和客户之间的交流提供共同认可的语言核心。随着项目的进展，领域模型不断被精化，最终成为整个软件的问题领域层，该层决定了软件系统能力的范围。本书还认为，从项目前期伊始，软件架构师就应该是领域建模活动的领导者，这样可以避免“不同阶段领域模型由不同人负责”所带来的问题。

接下来要进行概念性架构的设计。软件系统的规模越大、复杂程度越高，进行概念性架构设计的好处就越明显。

**确定对架构关键的需求。**这不仅要求对功能需求（如用例）进行筛选，还要对非功能需求进行综合权衡，最终确定对软件架构起关键作用的需求子集。也就是说，必须致力于“缩小范围”，这样就掌握了主动权，可以充分利用有限的时间把架构分析和架构设计工作做“深”，而不是碌碌无为地在多得数不清的需求之间徘徊。

**概念性架构设计。**设计概念性架构的第一步是分析关键用例的用例规约，运用鲁棒图构造系

统理想化的职责模型。接下来，明确架构模式，确定交互机制，形成初步的概念性架构。最后，还要通过质量属性分析，制定出满足非功能需求的高层设计决策，并根据这些设计决策对此前的工作成果进行增强、调整，以保证概念性架构体现这些设计决策。

在接下来，考虑具体技术的运用，设计出实际架构。

**细化软件架构。**此前的概念性架构所关注的关键设计要素、交互机制、高层设计决策多与具体技术无关，而最终的软件架构设计方案必须和具体技术结合，为开发人员提供足够的指导和限制。为此，我们必须从系统如何规划、如何开发、如何运行等角度揭示软件系统的结构和机制。一般而言，可以分别从逻辑架构、开发架构、运行架构、物理架构、数据架构等不同架构视图进行设计。

**验证软件架构。**对后续工作产生重大影响返工代价很高的任何工作都应该进行验证，软件需求如此，架构设计方案也是如此。至于验证架构的手段，对软件项目而言，往往需要开发出架构原型，并对原型进行测试和评审来达到；而对软件产品而言，可以开发一个框架（Framework）来贯彻架构设计方案，再通过在这个框架之上开发特定的垂直原型来验证特定的功能或质量属性。因此，从架构验证工作得到的不应该是“软件架构是否有效”的回答，还必须要有可实际运行的程序：体现软件架构的垂直抛弃原型或垂直演进原型，或者是更利于重用的框架。这些成果为后续的开发提供了实在的支持。

## 9.2 软件架构设计过程解析

### 9.2.1 架构设计策略应成为一等公民

我们都知道，计算机的 CPU 只实现加法运算就可以达到“计算完备”，将加减乘除统统作为用硬件实现的“一等公民”是为了实现高效。Java 继承并发扬了 C++，让接口成为“一等公民”，是为了促进 OO 思维。同样，尽量使架构设计策略作为软件架构设计过程的“一等公民”也将大有好处。

软件架构设计过程应该将（在第 7 章中有相关讨论）架构设计策略作为“一等公民”突出体现出来，只有这样，才能更有效地指导软件架构师进行架构设计。

图 9-3 展示了软件架构设计过程中对架构设计策略的具体体现。

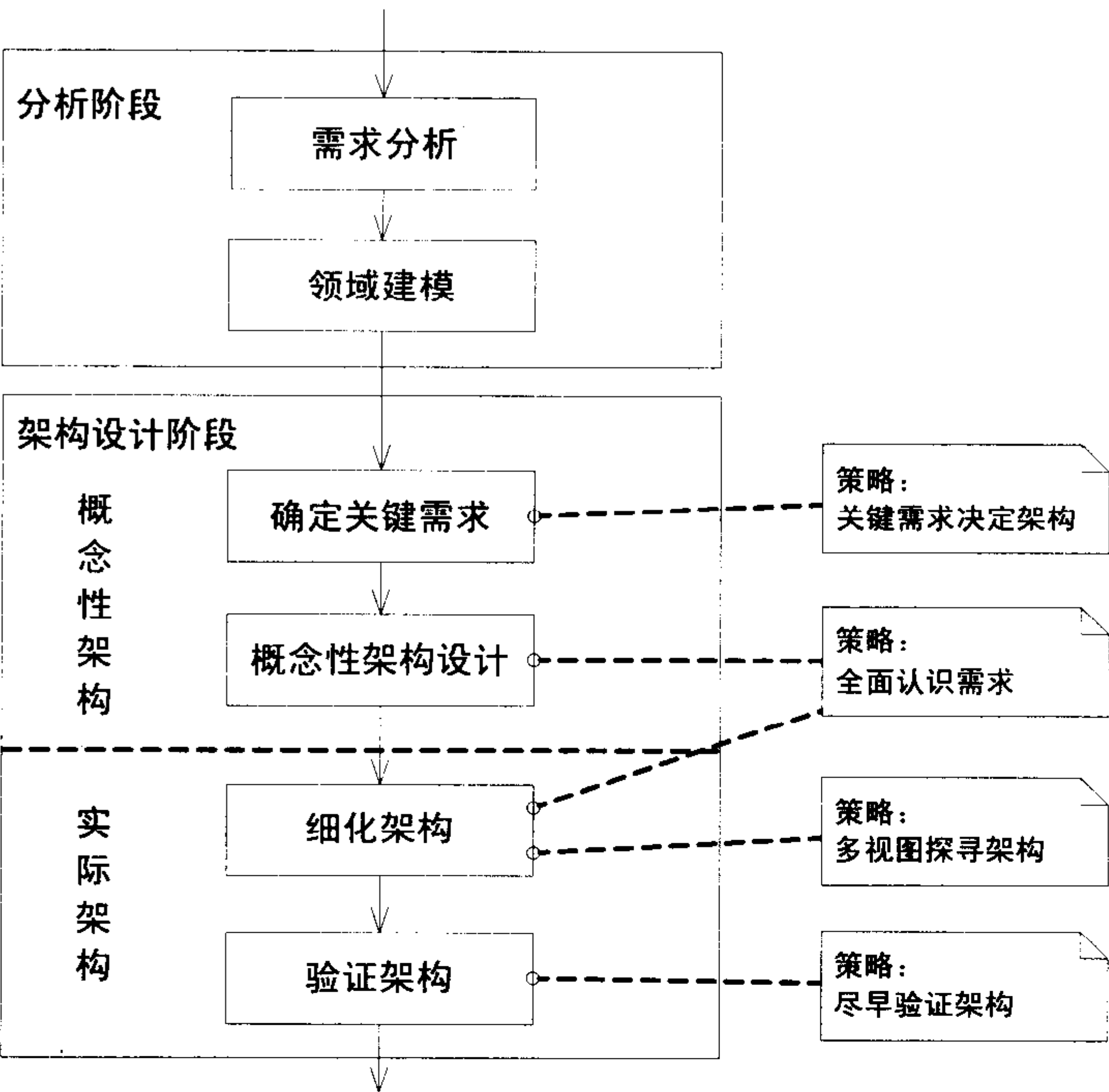


图 9-3 架构设计策略的体现

9.2.2 架构设计过程中的工作产品

著名的顾问和项目管理者 Stephen R. Palmer 说过：“好的过程清晰地定义任务。任务焦点放在结果上而不是细节上，项目可有效地进行，但仍然可以适应一些非常事件和环境的变化。”因此，本节从工作产品及工作产品之间的关系角度，来探讨软件架构设计过程。如图 9-4 所示。

领域模型凝聚了领域专家知识。问题领域可能很复杂，领域模型揭示了纷繁复杂的问题背后的结构。领域模型和软件需求不同：领域模型是对问题领域“做透视”，从而揭示其内在结构；而软件需求是对问题领域“拍照片”，从而捕捉其外在功能。领域模型是相对稳定的，而软件需求是变化的，一个优秀的领域模型可以“容纳”一定程度的需求变化。领域模型是团队交流的基础，是所有团队成员所使用语言的核心；当然，需求捕获和需求讨论时也应当使用领域模型规定的领域词汇，从而扫清业务人员和开发人员之间的交流障碍。



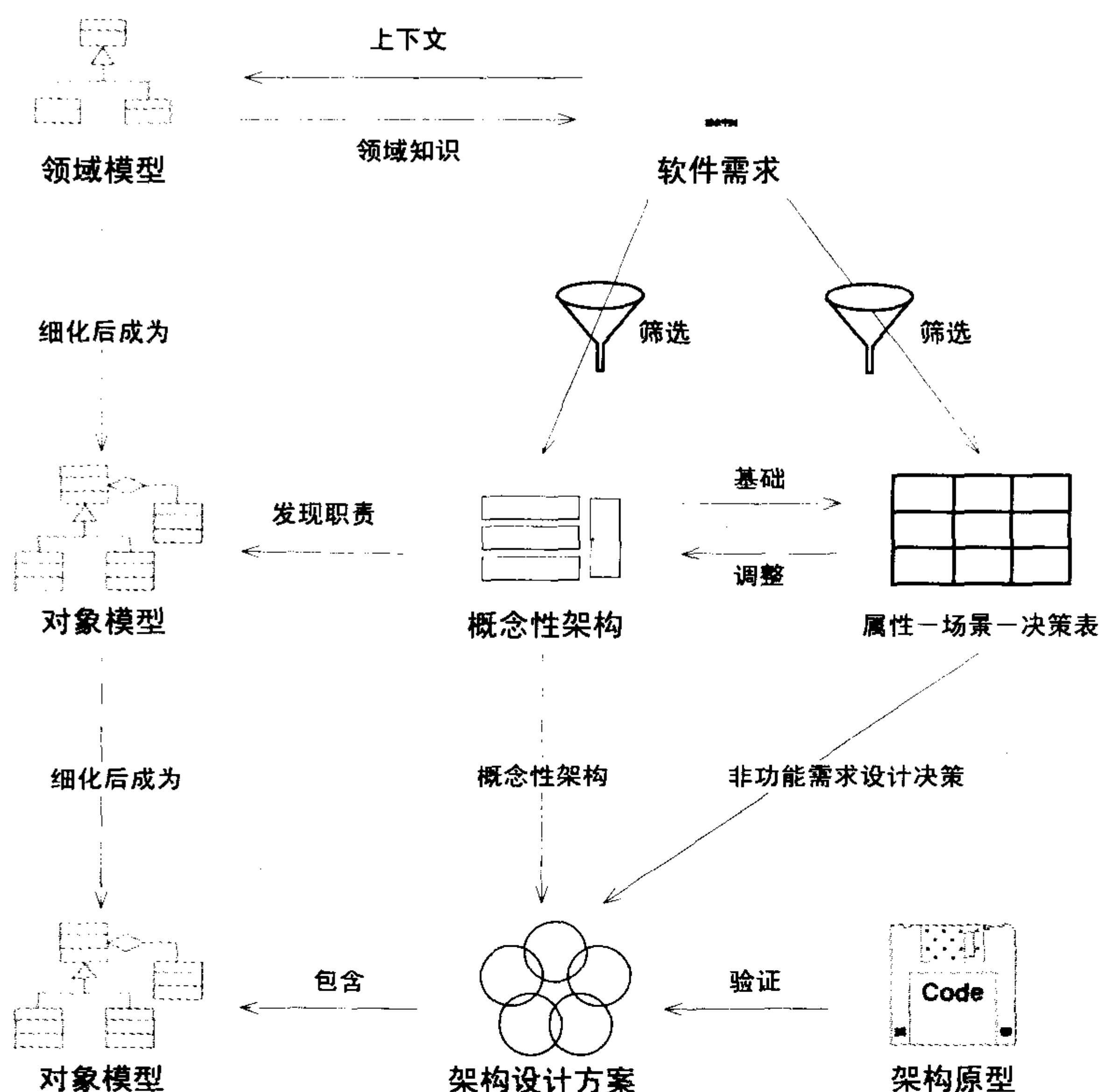


图 9-4 架构设计过程中的工作产品及其关系

软件需求是委托方希望软件系统达到的目标。软件需求不仅包括功能需求，还包括质量属性需求和约束性需求。全面认识需求对软件架构设计很重要，因为软件架构强调的是整体，而整体性的设计决策必须基于对需求的全面认识；另外，软件架构应该是稳定的，遗漏了重要需求的架构设计将面临返工的危险。软件需求必须验证。让最终用户参与到需求验证中来很关键，这时往往需要开发供最终用户体验和评审的软件原型。“基于原型的需求验证”在项目前期往往更像“需求启发”，因为用户会不断地提出这样或那样的意见。这很正常，就像 Norman 所说的，“用户直到看到不想要的东西的时候，才知道自己要什么”。

需求的数量很大，特别是功能需求。架构设计时考虑所有需求一般是不现实的，这不仅是因为时间压力，也是因为质量压力——在所有需求上平摊精力会导致我们的分析设计工作不够深

入。正如“关键需求决定架构”策略所提倡的，应该对需求进行权衡和筛选，找到对软件架构起关键作用的需求子集，让它们“驱动”下面的架构设计活动。

概念性架构是架构设计的初步成果。它通过识别主要的设计元素及它们之间的关系来描述系统。概念性架构也会确定软件系统所采用的架构模式。

其中，对软件架构起关键作用的质量属性需求将在“质量属性分析”活动中进行。正如我们将在第15章中看到的，为了制定满足质量属性的架构决策，可以采用“属性—场景—决策”表作为辅助思维的工具。

架构设计不断深入的过程，也是领域模型不断精化的过程。初期的领域模型或许已经明确了类的名称和属性，但类的方法往往没有明确。随着用例分析的进行，不断有类的职责被确定，作为类的方法被添加进来。

最终的软件架构设计方案必须和具体技术结合，从而为开发人员提供足够的指导和限制。当然，对架构设计进行细化时必须参照概念性架构，并遵循约束性需求规定的制约条件。基于多视图的架构设计方法被越来越多的人采用，一般而言，架构设计应针对如下几个方面进行：逻辑架构、开发架构、运行架构、物理架构、数据架构。

在设计软件的逻辑架构过程中，领域模型将被进一步精化，并成为软件逻辑架构的重要组成部分。一般而言，在此之前的领域模型并不区分“类”和“接口”，而是统统按“一般化的类”对待，而在此时的架构设计活动中将引入接口。

接下来就要通过具体实现技术来构造架构原型，对架构的有效性进行验证。架构原型可以是抛弃型的，也可以是演进型的；还可以先开发框架（Framework）来“固化”架构，然后基于框架开发很薄的抛弃原型来验证架构。

### 9.3 总结与强调

---

临石观海，可纵览全局。本章对软件架构设计过程进行总述。

从软件架构师的角度而言，他希望能有能够切实指导自己实际工作的架构设计过程。这倒不是说架构设计过程和整个软件开发过程有什么不一致的地方，而是说前者应当比后者有更多架构设计的“细节”。同时，软件架构设计过程应该将架构设计策略作为“一等公民”突出体现出来，只有这样，才能更有效地指导软件架构师进行架构设计。

本章是架构设计过程部分中提纲挈领的一章。如果说本章是立岸观海，那么下面各章就是入海航行了。

## 第 10 章 需求分析

---

第一，你必须弄清问题。第二，找出已知数与未知数之间的联系……

——波利亚，《怎样解题》

那些没有经验的问题解决者们，几乎无一例外，都是去匆忙地寻找解决办法，而不是先给要解决的问题下定义。

——杰拉尔德·温伯格，《你的灯亮着吗》

业内对架构的讨论仍沿用了传统思想：如果知道了系统需求，就可以为此系统构建架构。这种观点是缺乏远见的……

——Len Bass，《软件构架实践（第 2 版）》

软件架构师的工作很辛苦，还要面对“令人不悦”的需求变更。希望本章的内容能为软件架构师更有成效地工作提供一个突破口。

囫圇吞枣地认为“需求就是用户的要求”绝对是不够的。通过本章的讨论，你可以掌握需求的分类，了解每类需求是如何影响软件架构设计的。本立道生。参透了这一点，架构师就可能变紧迫为从容，分门别类地运用不同的分析和设计手段，最终设计出支持不同需求的架构方案。

需求变更常为我们的工作带来考验。但每类需求都会发生变更吗？每类需求的变更几率都一样大吗？通过本章的讨论，你可以了解到什么样的需求最容易变更，什么样的需求最不容易变更。参透了这一点，架构师就可以变被动为主动，选择具有代表性的且不易变的功能需求，再加上质量属性需求和约束性需求作为架构设计的驱动因素，来避免绝大多数功能需求变更对架构设计造成的冲击。

架构师必须满足来自所有涉众的需求吗？架构师总能够做到这一点吗？通过本章的讨论你可以了解到，对于架构设计而言，功能需求之间很少有矛盾或抵触关系（功能和模型有可能矛盾），但性能、鲁棒性和可重用性等许多质量属性之间却往往有制约或促进关系。参透了这一点，架构师就可以变局促为坦然，通盘考虑、折衷权衡，决定不同需求的满足程度，甚至决定在某些需求上不投入任何努力。

一言以蔽之，对需求分类、需求折衷和需求变更的研究，可能是架构师在运用自己丰富的技



术经验开始实际的架构设计之前，所需要进行的“第一项修炼”。

## 10.1 软件需求基础

### 10.1.1 什么是软件需求

什么是软件需求？简单地说，软件需求就是“这个软件到底要为用户做什么”。

IEEE 的软件工程标准术语表将需求定义为：

1. 用户所需的解决某个问题或达到某个目标所要具备的条件或能力。
2. 系统或系统组件为符合合同、标准、规范或其他正式文档而必须满足的条件或必须具备的能力。
3. 上述第一项或第二项中定义的条件和能力的文档表述。

而 RUP 是这样定义需求的：

需求描述了系统必须满足的情况或提供的能力，它就可以是直接来自客户需要，也可以来自合同、标准、规范或其他有正规约束力的文档（A requirement describes a condition or either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document.）。

### 10.1.2 需求捕获 vs. 需求分析 vs. 系统分析

需求捕获是获取知识的过程，知识从无到有、从少到多。需求采集者必须理解用户所从事的工作，并且了解用户和客户希望软件系统在哪些方面帮助他们。

需求分析是挖掘和整理知识的过程，它在已掌握知识的基础上进行。毕竟，初步捕获到的需求信息往往处于不同层次，也有一些主观甚至不正确的信息。而经过必要的需求分析工作之后，需求会更加系统、更加有条理、更加全面。

那么系统分析呢？如果说，需求分析致力于搞清楚软件系统要“做什么”的话，那么系统分析已经开始涉及“怎么做”的问题了。《系统分析》一书中写道：

简单地说，系统分析的意义如下：“系统分析是针对系统所要面临的问题，搜集相关的资料，以了解产生问题的原因所在，进而提出解决问题的方法与可行的逻辑方案，以满足系统的需求，实现预定的目标。”

需求捕获、需求分析以及系统分析之间的关系我们必须理解透彻，否则就会影响工作的有效进行。图 10-1 概括了三者之间的关系。

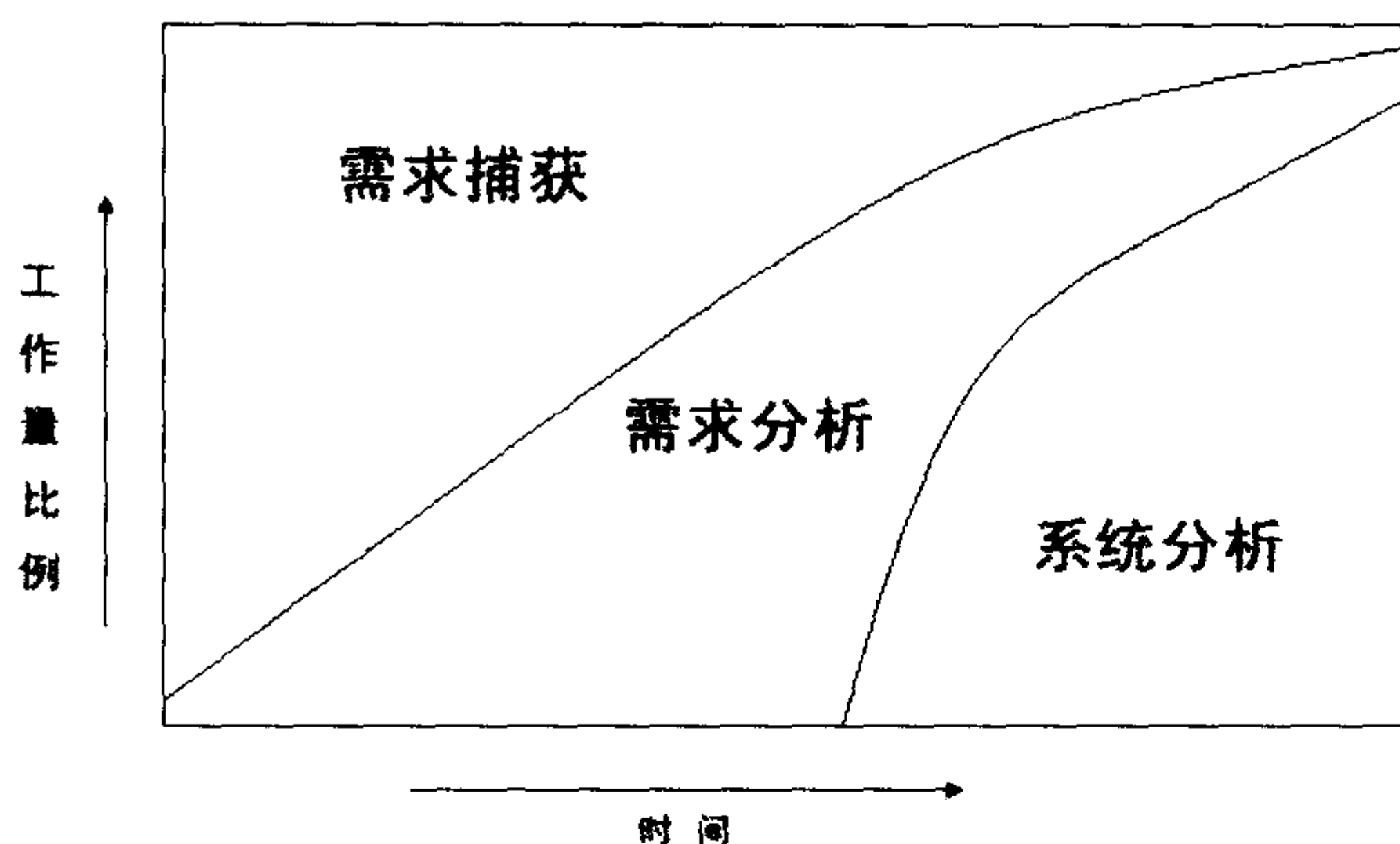


图 10-1 需求捕获、需求分析与系统分析

在实践中，需求分析和需求捕获的关系常常令人困惑。究其原因，是因为需求捕获、需求分析并不是先后进行的两个阶段性工作，相反，它们往往是相互伴随、交叉进行的：需求工作伊始，无疑更多的是进行需求捕获工作，相伴进行的需求分析工作占的比例偏少；但随着掌握的需求信息越来越多，我们需要开展的对需求的分析和整理工作也越来越多了。

同样，在实践中，需求分析和系统分析也常常被混淆。下面看看邵维忠教授和杨芙清院士在《面向对象的系统设计》中的精彩论述：

用“做什么”和“怎么做”来区分分析与设计，是从结构化方法沿袭过来的一种观点。但即使在结构化方法中这种说法也很勉强……

在“做什么”和“怎么做”的问题上为什么会出现上述矛盾？究其根源，在于人们对软件工程中“分析”这个术语的含义有着不同的理解——有时把它作为需求分析（Requirements Analysis）的简称，有时是指系统分析（Systems Analysis），有时则作为需求分析和系统分析的总称。

需求分析是软件工程学中的经典的术语之一，名副其实的含义应是对用户需求进行分析，旨在产生一份明确、规范的需求定义。从这个意义上讲“分析是解决做什么而不是解决怎么做的问题”是无可挑剔的。

但迄今为止人们所提出的各种分析方法（包括结构化分析和面向对象分析）中，真正属于需求分析的内容所占的分量并不太大；更多的内容是给出一种系统建模方法（包括一种表示法和相应的建模过程指导），告诉分析员如何建立一个能够满足（由需求定义所描述的）用户需求的系统模型。分析员大量的工作是对系统的应用领域进行调查和研究并抽象地表示这个系统。确切地讲，这些工作应该叫做系统分析，而不是需

求分析。它既是对“做什么”问题的进一步明确，也在相当程度上涉及到“怎么做”的问题。

忽略分析、需求分析和系统分析这些术语的不同含义，并在讨论中将它们随意替换，是造成上述矛盾的根源。

上述总结实在是鞭辟入里！首先是说明了需求分析致力于搞清楚软件系统要“做什么”，而系统分析更关注“怎么做”的问题。另外，也使我们明确了大多数分析方法（如 OOA）应该属于“系统分析”的范畴。

10.1.3 需求捕获及其工作成果

典型的需求捕获的工作成果是一摞“需求采集卡”（如表 10-1 所示），其中记录了需求类型、需求描述、需求背景、需求提出者和需求记录者等对进一步的需求分析非常有用的信息。

表 10-1 需求采集卡

需求采集卡					
项目		时间		地点	
需求类型		需求编号		用户关注度	
描述					
背景或原因					
相关材料					
提出者/受访者			记录者/采访者		

有的组织采用《需求调查报告》的形式，将采集到的需求进行归纳和汇总。在此不再赘述。

10.1.4 需求分析及其工作成果

通过需求采集活动，我们捕获到了大量“原始需求”，而需求分析则对采集到的原始需求进行分析、整理、辨别和归纳，最终形成系统的、明确的软件需求。

何为需求分析？需求分析就是对用户需求进行分析，以得到一份明确的、规范的需求定义。

需求分析的工作成果是《软件需求规格说明书》（Software Requirements Specification, SRS），它精确地阐述了一个软件系统必须提供的功能、必须达到的质量属性指标以及它必须遵守的约束。SRS 应尽可能完整地描述各种条件下的系统行为。



SRS 通常包括如下内容:

1. 前言
  - a) 目的
  - b) 范围
  - c) 定义、缩写词、略语
  - d) 参考资料
2. 需求概述
  - a) 用例模型
  - b) 限制与假设
3. 具体需求
  - a) 用例描述
  - b) 外部接口需求
    - i. 用户接口
    - ii. 硬件接口
    - iii. 软件接口
    - iv. 通信接口
  - c) 质量属性需求
    - i. 性能
    - ii. 易用性
    - iii. 安全性
    - iv. 可维护性……
  - d) 设计和实现约束
    - i. 必须遵循的标准
    - ii. 硬件的限制……

### 10.1.5 系统分析及其工作成果

需求分析致力于搞清楚软件系统要“做什么”，而系统分析已经开始关注“怎么做”的问题。

对于不同的系统分析方法，其工作成果差异很大。通过结构化分析方法得到的最重要的工作成果是数据流图；而面向对象的系统分析方法得到的工作成果主要是分析类图、鲁棒图、序列图等——其中分析类图描述设计的静态方面，而鲁棒图和序列图描述设计的动态方面。

## 10.2 需求分析在软件过程中所处的位置

本章要讨论的需求分析活动，是整个分析阶段的重要工作之一。

### 10.2.1 概念化阶段所做的工作

在分析阶段之前，是概念化阶段。概念化阶段要解决项目的起源问题，主要针对项目目标、主要特性、功能范围和成功要素等进行构思并达成一致。

概念化阶段最重要的工作成果是《愿景与范围文档》。记得有一位大师，当被问及“如果软件开发中只能有一份文档，应当是哪一份”时，他毫不犹豫地回答说，是《愿景文档》。由此可见《愿景与范围文档》的重要性。

从事软件开发的有很多种类型的组织，例如项目型公司、产品型公司、外包型公司、服务型公司等。因此，不同的组织，《愿景与范围文档》可能有着不同的名字。例如，有的产品型公司将《愿景与范围文档》称为《市场需求文档》（Market Requirements Document, MRD）。

典型的《愿景与范围文档》包括下列内容：

1. 业务需求
  - a) 目背景
  - b) 业务机遇
  - c) 业务目标
  - d) 客户或市场需求
  - e) 提供给客户的价值
  - f) 业务风险
2. 项目愿景的解决方案
  - a) 目项目愿景陈述
  - b) 主要特征
  - c) 假设和依赖环境
3. 范围和局限性
  - a) 目首次发布的范围
  - b) 随后发布的范围
  - c) 局限性和专用性
4. 业务环境

- a) 目客户概貌

b) 项目的优先级

5. 产品成功的因素

在《愿景与范围文档》中，上下文图（Context Diagram）扮演着重要角色，它清晰地描述了待开发系统与周围所有事物之间的界限与联系。由此，可以明确哪些方面在软件系统内部、哪些方面在软件系统之外、谁使用该系统以及此软件系统需要和哪些相关软件系统进行交互等问题。一般而言，上下文图是由市场部门绘制和管理的，所采用的画法相对随意。当然也可以用 UML 来绘制上下文图，图 10-2 就是一个银行系统的例子。

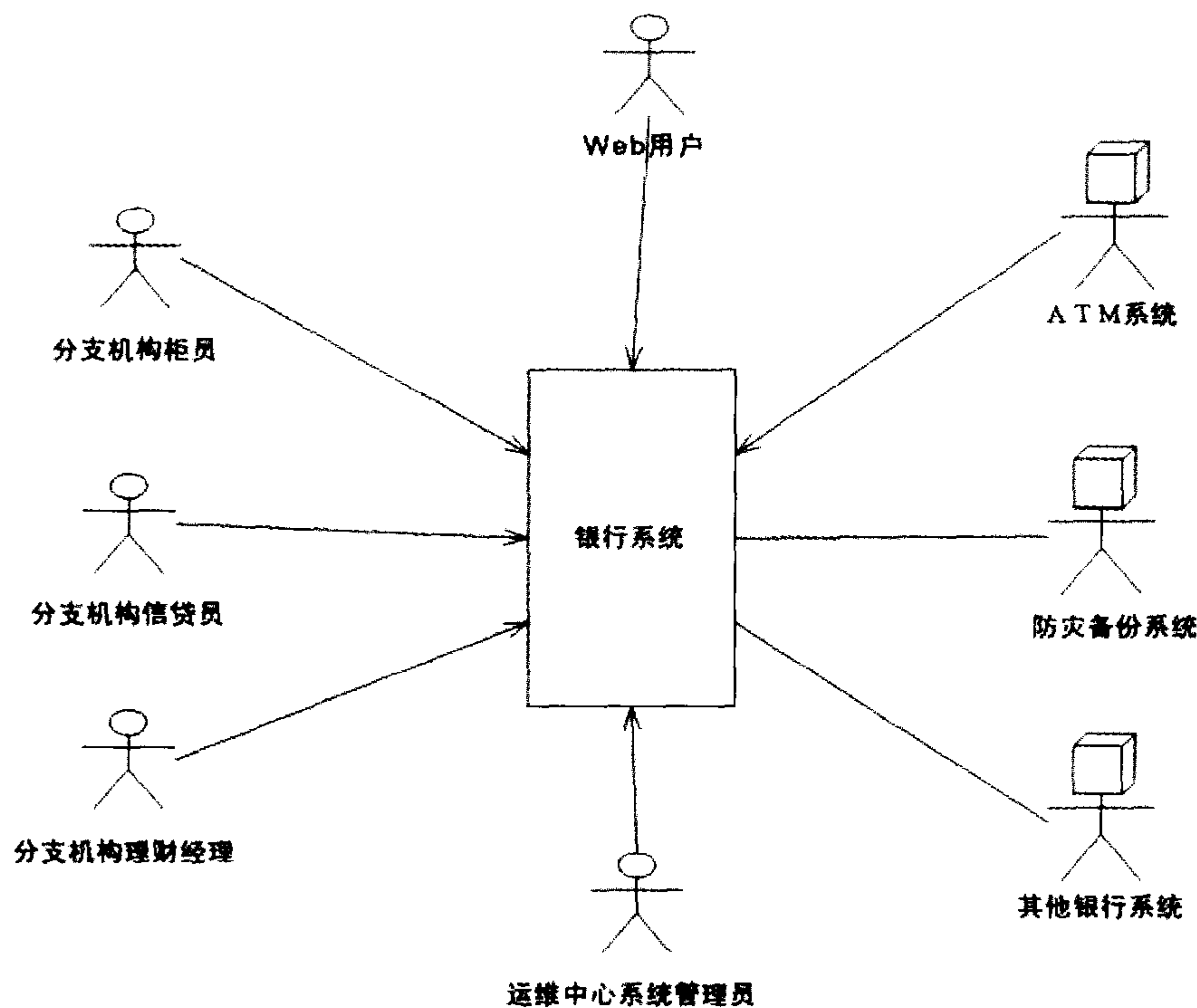


图 10-2 银行系统的上下文图

一般而言，对定制开发的软件项目来说，概念化阶段需要需方的高层领导参与，明确业务需要，由需方自行组织人员阐述业务需求；而对产品型的软件公司来说，在概念化阶段，往往是产品管理部门根据市场部门的要求，定义产品所要提供的业务功能。无论上述哪种情况，都应阐明业务需求的需求描述及该需求产生的背景和理由等。

当然，概念化阶段还需进行风险评估、可行性分析以及项目进度和成本的粗略预估等工作。为了就项目愿景达成确切的共识（而不是假设的共识），应当开发一个抛弃型原型（对原型技术



的系统介绍请参考第 17 章); 通过基于原型的评估还有助于发现项目风险和估算项目规模。

### 10.2.2 需求分析所处的位置

概念化阶段的工作为需求分析奠定了基础、划定了范围, 接下来就可以进行需求分析了 (如图 10-3 所示)。

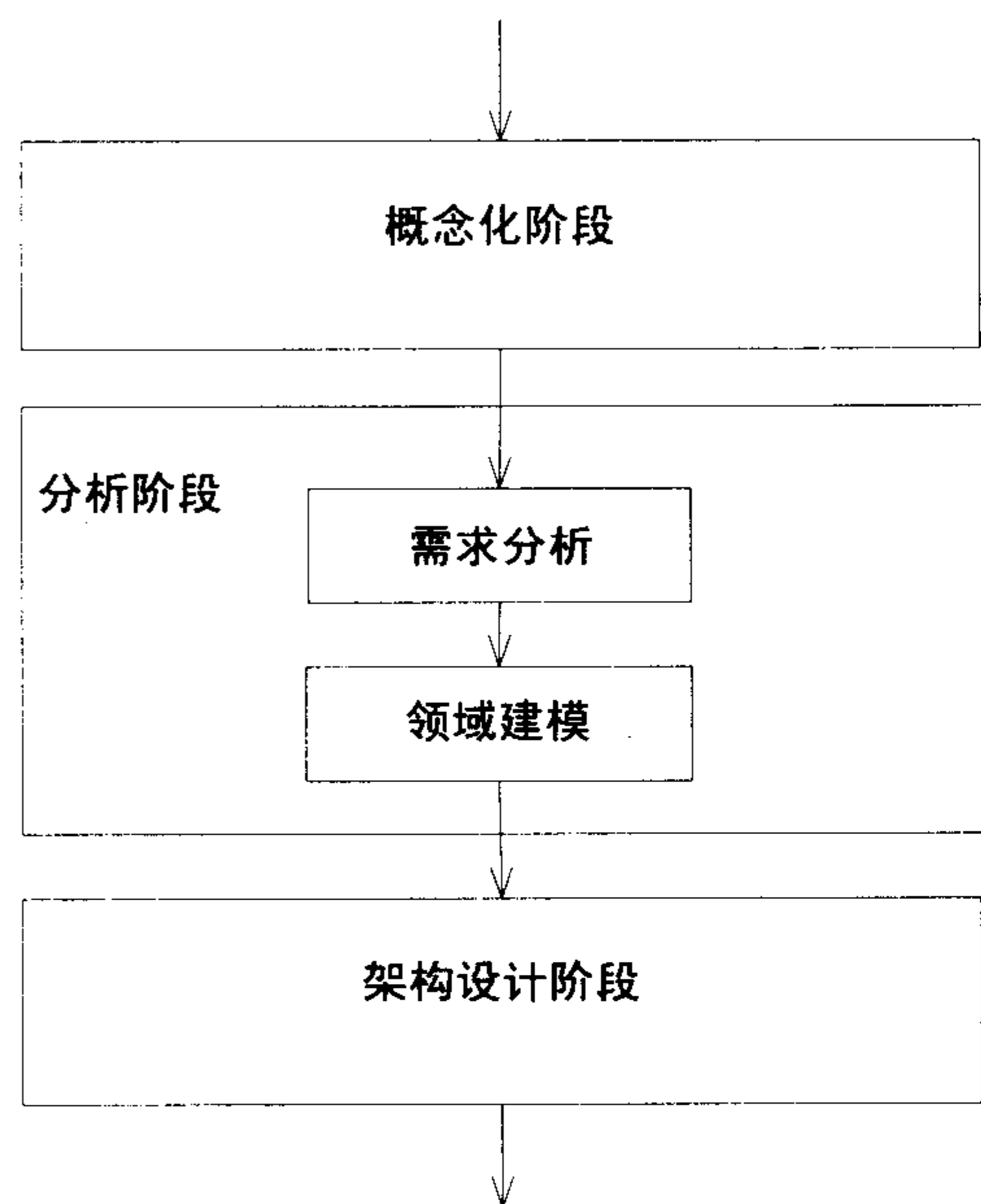


图 10-3 需求分析活动在过程中的位置

概念化阶段明确了软件项目的意义、可行性及概况等, 在《愿景与范围文档》所划定的大前提的指导下, 需求分析活动要进一步完善和细化软件需求。

需求分析和领域建模是相互支持的关系。不难理解, 要进行领域建模, 很大程度上依赖于需求讨论会等活动。同时, 领域模型作为领域建模的成果, 规定了重要的领域词汇表, 并且这些词汇的定义是严格的、大家共同认可的, 所以可以成为团队交流的基础, 自然也应当作为需求分析活动和《软件需求规格说明书》应当遵循的标准词汇。

对于后期的架构设计活动而言, 需求分析活动应该提供功能需求、质量属性需求以及约束性需求等不同需求的明确定义。软件架构师将就此开展用例分析 (当功能需求采用用例描述时)、质量属性分析、细化架构设计等活动。

# 10.3 架构师必须掌握的需求知识

软件架构师不必是需求捕获专家，也不必是编写《软件需求规格说明书》的专家。但他一定应在需求分类、需求折衷和需求变更的研究方面是专家，否则他和其他软件架构师相比，就输在了“起跑线”上。

## 10.3.1 软件需求的类型

以工程领域的例子做个类比吧。

比如设计一座跨江大桥：

- 我们会考虑“连接南北的公路交通”这个“功能需求”，从而初步设计出理想化的桥墩支撑的公路桥方案；
- 然后还要考虑造桥要面临的“约束条件”，这个约束条件可能是“不能影响万吨轮从桥下通过”，于是细化设计方案，规定桥墩的高度和桥墩之间的间距；
- 另外还要顾及“大桥的使用期质量属性”，比如为了“能在湍急的江流中保持稳固”，可以把大桥桥墩深深地建在岩石层之上，和大地浑然一体；
- 其实，“建造期间的质量属性”也很值得考虑，比如在大桥的设计过程中的一些考虑“施工方便性”的措施。

和工程领域的功能需求、约束条件、使用期质量属性、建造期间的质量属性等类似，软件系统的需求种类也相当复杂，具体分类如图 10-4 所示。

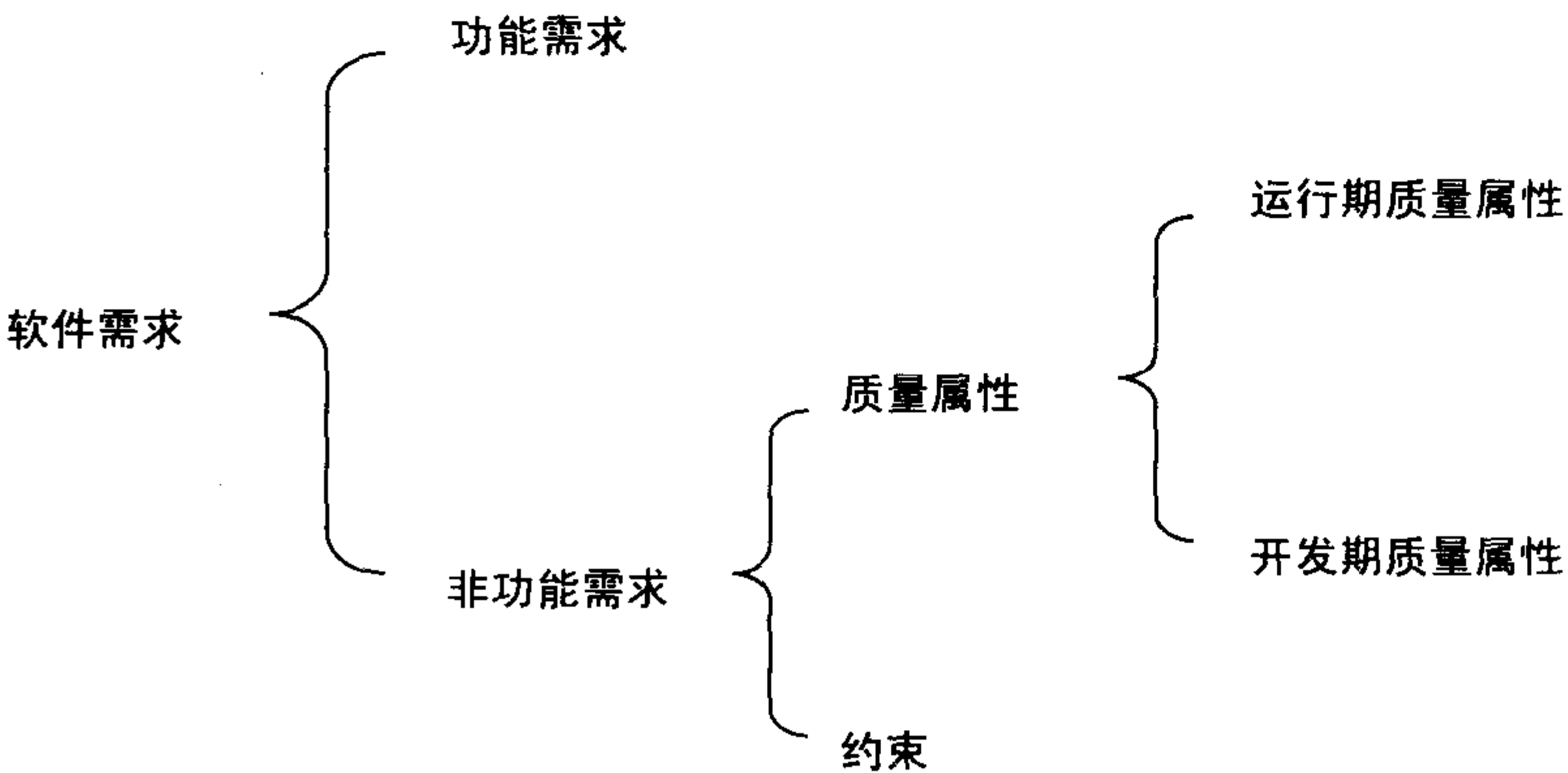


图 10-4 软件需求的类型

功能需求是我们最熟悉的一类需求。功能需求描述要开发的软件系统应该做什么，它可以通过“软件系统应提供的服务”的形式来定义，既包括为用户提供的服务，又包括本系统为其他系统提供的服务。

非功能需求中有一类称为约束的需求，它规定了开发软件系统时必须遵守的限制条件。例如，采用何种操作系统、采用何种开发技术、需要和哪些遗留系统进行互操作等，可以视为技术性约束。而为了获得相关行业或组织的认可，或者大型企业集团处于长期整合规划的要求，软件的设计和开发可能还必须遵守相关的行业标准、企业标准等标准的约束。当然，根据具体情况的不同，还可能需遵守相关的法律、法规、政府规章等行政或法律约束。上述情况均可视为约束性需求这一大类。

功能需求强调行为，而约束不是行为。Craig Larman 在《UML 和模式应用（第 2 版）》写道：

**约束**不是行为，是设计或项目的某些限制条件。这些限制条件也属于需求，但通常被称为“约束”来强调其限制性。例如：

- 必须使用 Oracle（我们硬件签署过使用许可证了）；
- 必须在 Linux 上运行（成本低）。

下面重点讨论非功能需求中的质量属性需求。

如何给众多质量属性分类，这是一个问题。McCall 等人于 1977 年提出的软件质量属性的分类模型，影响非常广泛。如图 10-5 所示，它将软件的质量属性划分为三大类：产品操作、产品修改、产品改型。

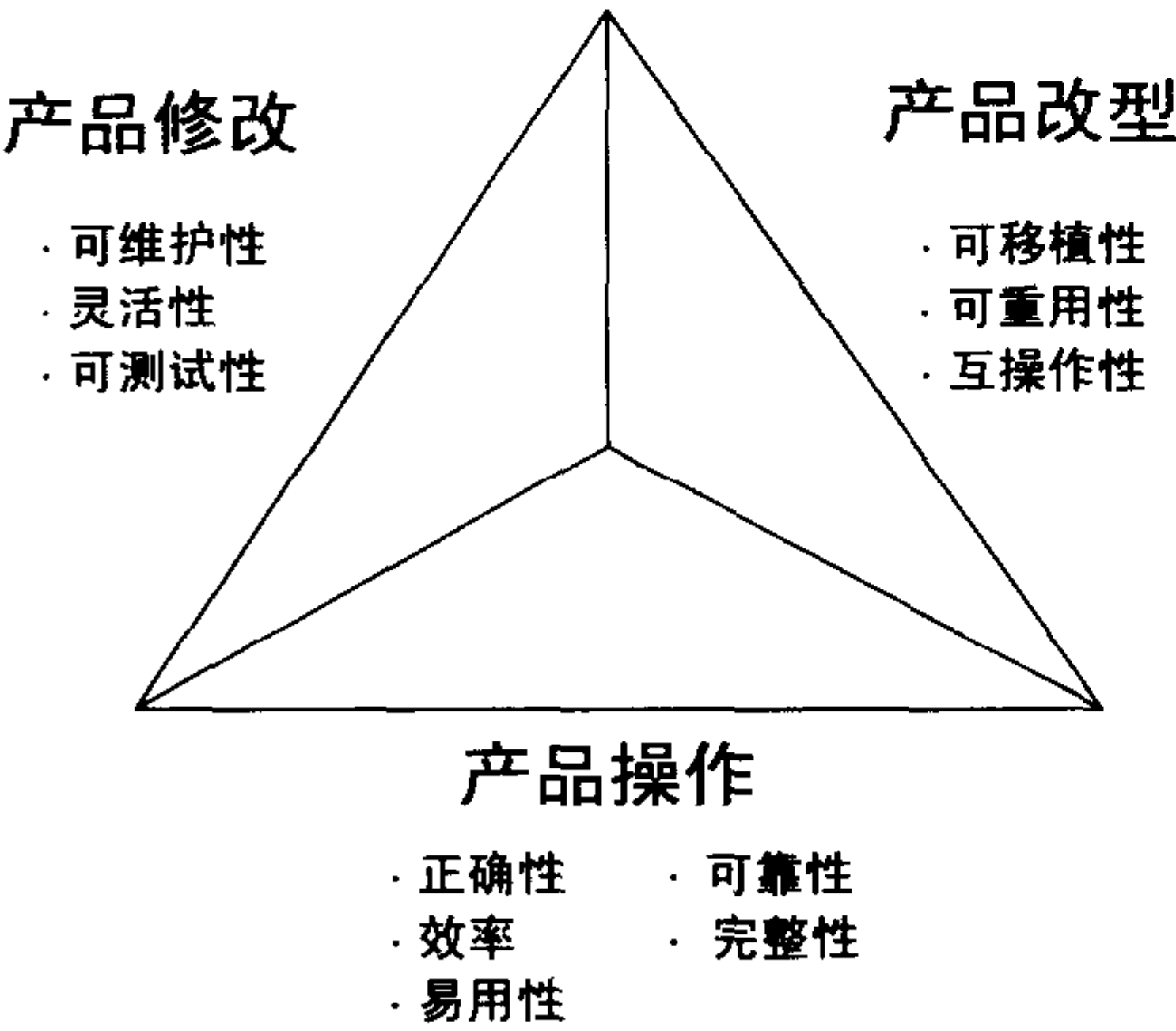


图 10-5 McCall 等人提出的软件质量属性的分类模型



该分类方法相当经典，但似乎缺少了“产品开发”类的质量属性——诸如易理解性、可扩展性和可重用性等。另一方面，我们也发现产品开发、产品修改和产品改型这三类质量属性有重叠（如易理解性和可扩展性对开发和维护都很重要，再如对产品改型很关键的可重用性也同样影响着产品开发）。因此，考虑到被广泛认同的迭代式开发所暗示的增量交付使开发、修改和维护之间的界限不再像以前那么明显，所以本书认为有充分的理由将产品开发、产品修改和产品改型这三类质量属性合并为一类。

本书推荐将软件质量属性划分为运行期质量属性和开发期质量属性两大类（如表 10-2 所示）：

- 开发期质量属性其实包含了和软件开发、维护和移植这三类活动相关的所有质量属性，可以说这里的“开发”是相当广义的；
- 开发期质量属性是开发人员、开发管理人员和维护人员都非常关心的，对最终用户而言，这些质量属性只是间接地促进用户需求的满足；
- 运行期质量属性是软件系统在运行期间，最终用户可以直接感受到的一类属性，这些质量属性直接影响着用户对软件产品的满意度。

表 10-2 推荐的软件质量属性分类方式

运行期质量属性	开发期质量属性
性能（Performance） 安全性（Security） 易用性（Usability） 持续可用性（Availability） 可伸缩性（Scalability） 互操作性（Interoperability） 可靠性（Reliability） 鲁棒性（Robustness）	易理解性（Understandability） 可扩展性（Extensibility） 可重用性（Reusability） 可测试性（Testability） 可维护性（Maintainability） 可移植性（Portability）

运行期质量属性需求是一类非常重要的非功能需求，对客户满意度非常关键，下面一一进行说明。

**性能（Performance）。**性能是指软件系统及时提供相应服务的能力。具体而言，性能包括速度、吞吐量和持续高速性三方面的要求：

- 吞吐量通过单位时间处理的交易数来度量；
- 速度往往通过平均响应时间来度量；
- 而持续高速性是指保持高速处理速度的能力。

持续高速性和实时系统有关，实时系统有“硬实时”和“软实时”之分，其中硬实时系统对



每次系统响应时间都有严格要求，如果不能满足要求，后果将是致命的，所以把性能笼统地说成“进行典型操作所需的时间”显然忽视了实时系统性能的内涵。

下面值得说明一下效率（Efficiency）和性能的关系：它们反映了同一问题的“表”、“里”两面，性能为“表”，效率为“里”，如图 10-6 所示。所谓效率，是指软件系统对 CPU 处理能力和存储能力这两大类计算机资源的使用效率。

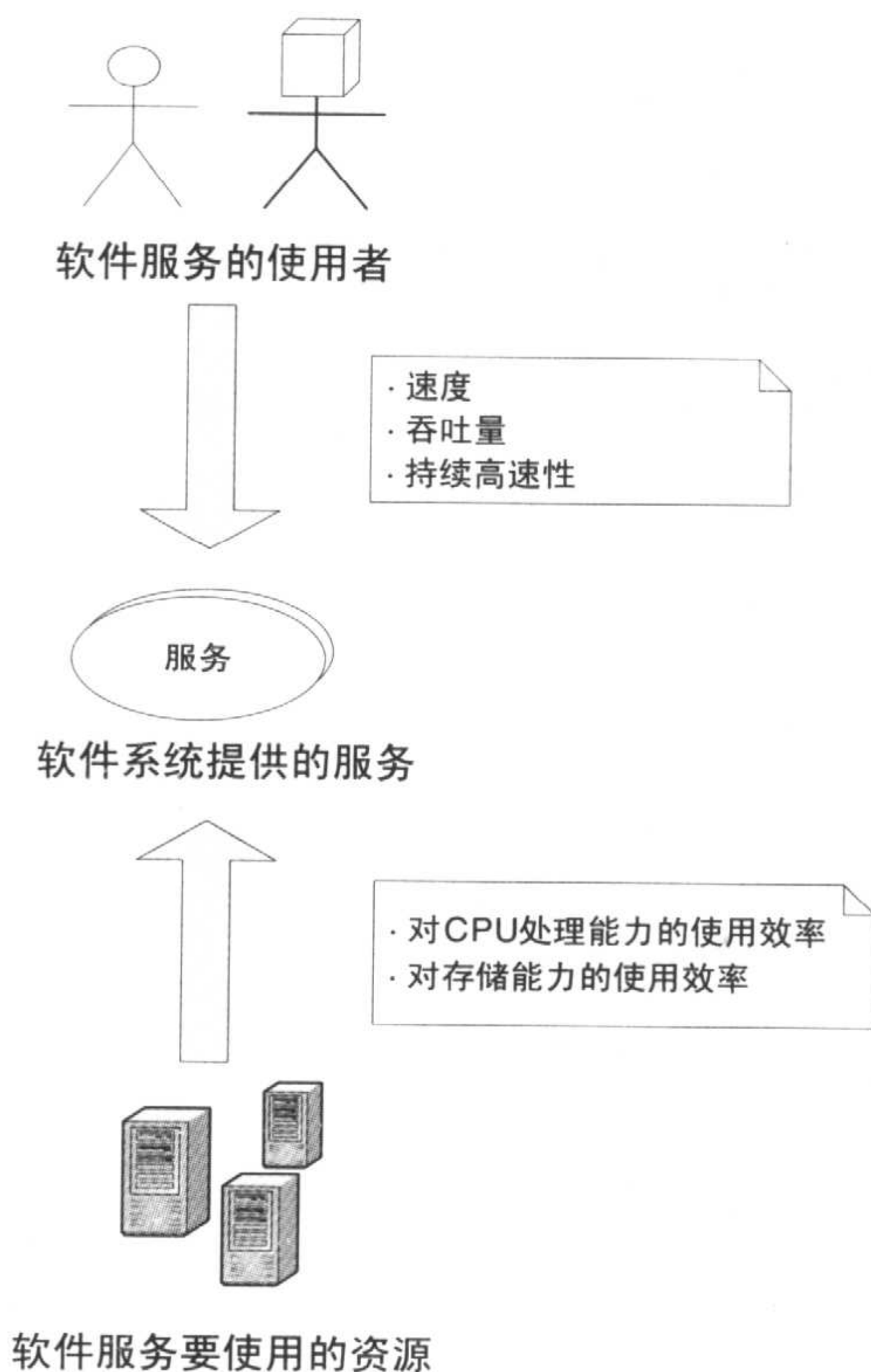


图 10-6 性能和效率的关系

**安全性（Security）。**指软件系统同时兼顾向合法用户提供服务，以及阻止非授权使用的能力。高安全性意味着“同时兼顾”，这是因为有些攻击的目的是使软件系统拒绝向合法用户提供服务，而不是非法访问。

**易用性（Usability）。**不少文献也称之为可用性，但为了避免和持续可用性（Availability）混淆，本书采用非常流行的“易用性”的叫法。指软件系统易于使用的程度。

**持续可用性（Availability）。**不少文献也称之为可用性，但为了避免和易用性（Usability）混淆，本书采用“持续可用性”的叫法。持续可用性指系统长时间无故障运行的能力。

**可伸缩性 (Scalability)**。指当用户数和数据量增加时，软件系统维持高服务质量的能力。例如当业务量较小时，软件系统运行在一台服务器上，当业务量增大时，可以通过增加服务器或增加单台服务器上所运行软件系统的个数来提高性能，而无需对软件系统本身进行编程级的修改。

**互操作性 (Interoperability)**。指本软件系统与其他系统交换数据和相互调用服务的难易程度。

**可靠性 (Reliability)**。软件系统在一定的时间内无故障运行的能力。

**鲁棒性 (Robustness)**。也称健壮性、容错性。鲁棒性是指软件系统在以下情况下仍能够正常运行的能力：用户进行了非法操作；相连的软硬件系统发生了故障，以及其他非正常情况。

而开发期质量属性则随着软件系统规模的日益增长，显得越来越重要了，下面一一说明之。

**易理解性 (Understandability)**。尤指设计被开发人员理解的难易程度。

**可扩展性 (Extensibility)**。为适应新需求或需求的变化为软件增加功能的能力。我们在实际工作中，经常将可扩展性称为灵活性。

**可重用性 (Reusability)**。重用软件系统或其一部分的能力的难易程度。

**可测试性 (Testability)**。对软件测试以证明其满足需求规约的难易程度。在实际工作中主要指进行单元测试、插桩测试等的难易程度。

**可维护性 (Maintainability)**。为了达到下列三种目的之一，而定位修改点并实施修改的难易程度：修改 Bug、增加功能、提高质量属性。

**可移植性 (Portability)**。将软件系统从一个运行环境转移到另一个不同的运行环境的难易程度。

务实地，我们可以将运行期质量属性和功能性一起视为“软件的外部质量”，而将开发期质量属性视为“软件的内部质量”。无疑，软件的内部质量制约着软件的外部质量；在软件开发管理本身已经十分复杂的今天，想使内部质量很差的软件具有良好的外部质量几乎是不可能的。同时，随着商业环境变化的加剧，很多企业软件出现了“建成即废弃”的尴尬情况。于是，软件系统的内部品质越来越受到重视，通过强化软件系统的可扩展性、可重用性、易理解性等开发期质量属性，可以使软件有更多被改变、被重用的空间。

### 10.3.2 各类需求对架构设计的影响

一个成功的软件架构师，不会对所有需求“胡子眉毛一把抓”，而是分门别类梳理清楚，不同的需求对架构设计的作用方式不同。图 10-7 总结了各类需求与软件架构之间的关系。



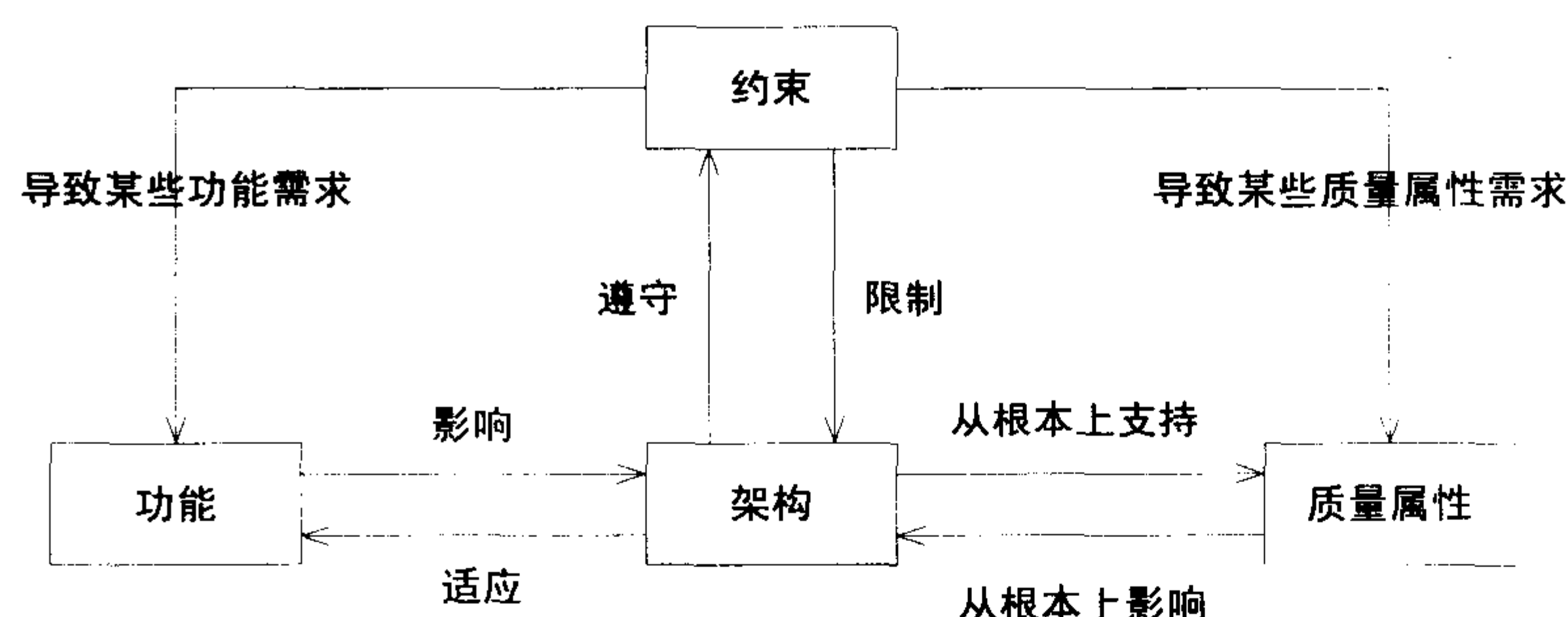


图 10-7 各类需求与软件架构之间的关系

功能需求影响架构，而架构必须适应功能需求。但功能需求并不能决定架构，这是显而易见的——因为如果仅为了满足功能需求而进行架构设计，那么设计结果几乎是毫无约束的，基于接口编程还是统统硬编码到实现都能实现功能需求，分不分层，以及如何分层似乎也无不同。

倒是质量属性需求对软件架构影响更大。例如，为了获得高可移植性，架构设计中必须考虑对硬件和平台相关特性进行封装和隔离。再例如，精心规划职责模型是获得高性能的根本，这就是为什么“将性能放在首位的软件系统”有时其架构与众不同的原因。

反过来，大部分质量属性需求能否被满足，也很大程度上依靠软件架构的设计。例如性能、可扩展性和可测试性等，虽然也受到编码质量的影响，但架构设计是否合理更为关键。再例如，微内核架构对可扩展性、可重用性和可移植性等有极大提升，适用作为生存期非常长（如 20 年）的软件系统的架构，以应对源源不断的变化；但微内核架构降低了软件系统的易理解性，增加了系统的复杂性。

约束性需求最为特殊，它可能产生的影响有 3 种：

- 作为架构设计时必须遵守的限制条件（例如“必须运行于 Linux 平台”）；
- 导致软件系统必须提供某些功能需求（参见图 10-8 之例）；
- 导致软件系统必须满足某些约束性需求（参见图 10-8 之例）。

非功能需求			功能需求
约束	运行期质量属性	开发期质量属性	
必须执行国家统一规定的利率，并与最新公布的利率调整方案保持一致	可配置性		调整利率的实用功能
.....	.....	.....	.....

图 10-8 从约束性需求导出功能需求和质量属性需求的例子

图 10-8 展示了对某银行系统进行需求分析的片断。利息率是国家统一规定的，并且利率调整方案未必提前公布，所以对银行系统所有利率的调整必须能在 1 小时之内迅速完成。由此，银行系统应具有良好的可配置性，并提供相应的调整利率的实用功能。

总之，全面正确地对需求分类，为后续工作奠定了格局基础，是整个项目成功的第一步。

### 10.3.3 超市系统案例：领会需求类型的不同影响

例子是最好的老师。

为了更好地理解不同种类软件需求的不同影响，我们来分析一个实际的例子。在表 10-3 中，我们列举了一个典型的超市系统的需求子集。

表 10-3 超市系统案例：理解需求种类

非功能需求			功能需求
约束	运行期质量属性	开发期质量属性	
项目预算有限 用户的平均电脑操作水平偏低 要求能在 Linux 上运行 开发人员分散在不同地点 .....	高性能 易用性 .....	易理解性 模块间松散耦合 .....	提高收银效率 任意商品项可单独取消 通过收银终端的按键组合，可以使收银过程从“逐项录入状态”进入“选择取消状态” .....

从这个例子中可以清晰地看到需求可以分为两大类：功能需求和非功能需求。

简而言之，功能需求就是“软件有什么用，软件需要做什么”。同时，注意把握功能需求的层次性是软件需求的最佳实践。以该超市系统为例：

- 超市老板希望通过软件来“提高收银效率”；
- 那么，你可能需要为收银员提供一系列功能来促成这个目的，比如供收银员使用的“任意商品项可单独取消”功能有利于提高收银效率（笔者曾在超市有过被迫整单取消然后一车商品重新扫描收费的痛苦经历）；
- 而具体到这个超市系统，系统分析员可能会决定要提供的具体功能为：通过收银终端的按键组合，可以使收银过程从“逐项录入状态”进入“选择取消状态”，从而取消某项商品。

从上面的例子中我们还惊讶地发现，非功能需求——人们最经常忽视的一大类需求——包括的内容非常多，并且极其重要。非功能需求又可以分为如下三类：

- 约束。要开发出用户满意的软件并不是件容易的事，而全面理解要设计的软件系统所



面临的约束可以使你向成功迈进一步。约束性需求既包括企业级的商业考虑（例如“项目预算有限”），也包括最终用户级的实际情况（例如“用户的平均电脑操作水平偏低”）；既可能包括具体技术的明确要求（例如“要求能在 Linux 上运行”），又可能需要考虑开发团队的真实状况（例如“开发人员分散在不同地点”）。这些约束性需求当然对架构设计影响很大，比如受到“项目预算有限”的限制，架构师就不应选择高技术成本的手段或昂贵的中间件等，而考虑到“开发人员分散在不同地点”，就更应注重软件模块职责划分的合理性和松耦合性等；

- 运行期质量属性。这类需求主要指软件系统在运行期间表现出的质量水平。运行期质量属性非常关键，因为它们直接影响着客户对软件系统的满意度，大多数客户也不会接受运行期质量属性拙劣的软件系统。常见的运行期质量属性包括软件系统的易用性、性能、可伸缩性、持续可用性、鲁棒性、安全性等。在我们的超市系统的案例中，用户对高性能提出了具体要求（真正的性能需求应该量化，我们的表 10-2 没体现），他们不能容忍金额合计超过 2 s 的延时；
- 开发期质量属性。这类非功能需求中的某些项人们倒是念念不忘，可惜很多人并没有意识到“开发期质量属性”和“运行期质量属性”对架构设计的影响到底有何不同。开发期质量属性是开发人员最为关心的，要达到怎样的目标应根据项目的具体情况而定，而过度设计（Overengineering）会付出额外的代价。

#### 10.3.4 各类需求的“易变更性”不同

需求的变更可以说“让我欢喜让我忧”——其中既蕴藏了风险又包含了机遇。

之所以说需求变更蕴含着风险，是因为不存在不需要成本的需求变更。任何需求变更都可能意味着时间和金钱的消耗，并且大量需求变更之后程序可能被搞得混乱不堪，而内部质量降低势必造成外部质量下滑，例如 Bug 增多等。

当然，需求变更也蕴含着机遇。对软件架构设计而言，这个机遇可能意味着设计出稳定的架构，最终这个架构能够支持业务功能在一定范围内“按需应变”。

需求为什么会变更呢？总结而言，需求变更可能有三类来源（如图 10-9 所示）：

- 我们要解决的问题发生了变化。商业环境变了（如行业软件）、国家政策变了（如电子政务系统）、用户兴趣变了（如游戏软件）都会引起我们要解决的问题发生变化；
- 我们对问题的理解发生了变化。客户和用户对他们要解决的问题的认识会改变。同样，开发团队对需求的理解也会不断加深。有些没有经验的开发人员单纯地认为客户提的需求不够确切，显然是忽视了“需求理解”其实是一个包含客户和开发方都在内的“信息传递链”这一事实；
- 我们理解问题的过程有误。例如我们了解需求找错了人。例如负责采集和分析需求的



需求分析员的技能拙劣。

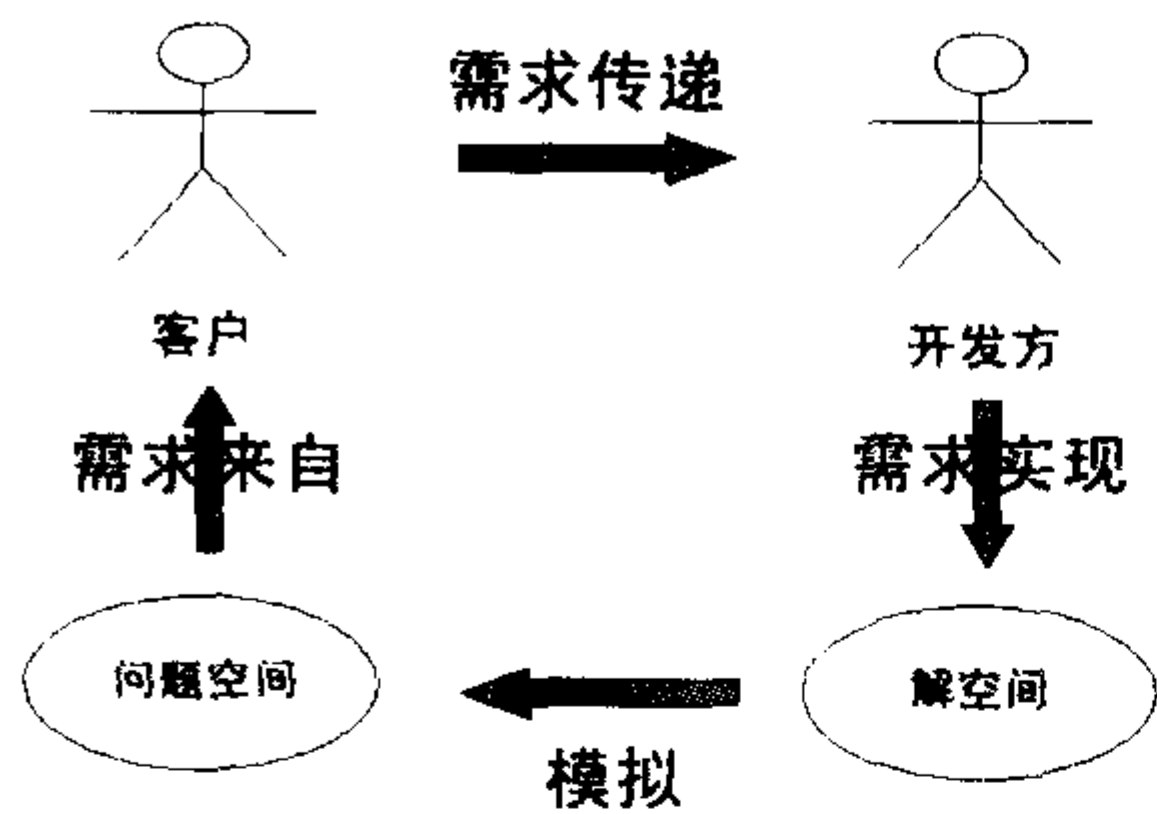


图 10-9 软件需求信息传递链

值得庆幸的是，大量经验表明，一个软件系统的各类需求“易变化性”并不相同。或者反过来，不同种类的需求的稳定性并不相同，一刀切地认为“需求变更无处不在”并不正确，甚至导致放弃了设计更稳定的软件架构的机会。着实可惜。

关于需求变更是有规律可循的。一般而言，功能需求最易变，而质量属性需求最稳定（如图 10-10 所示）。

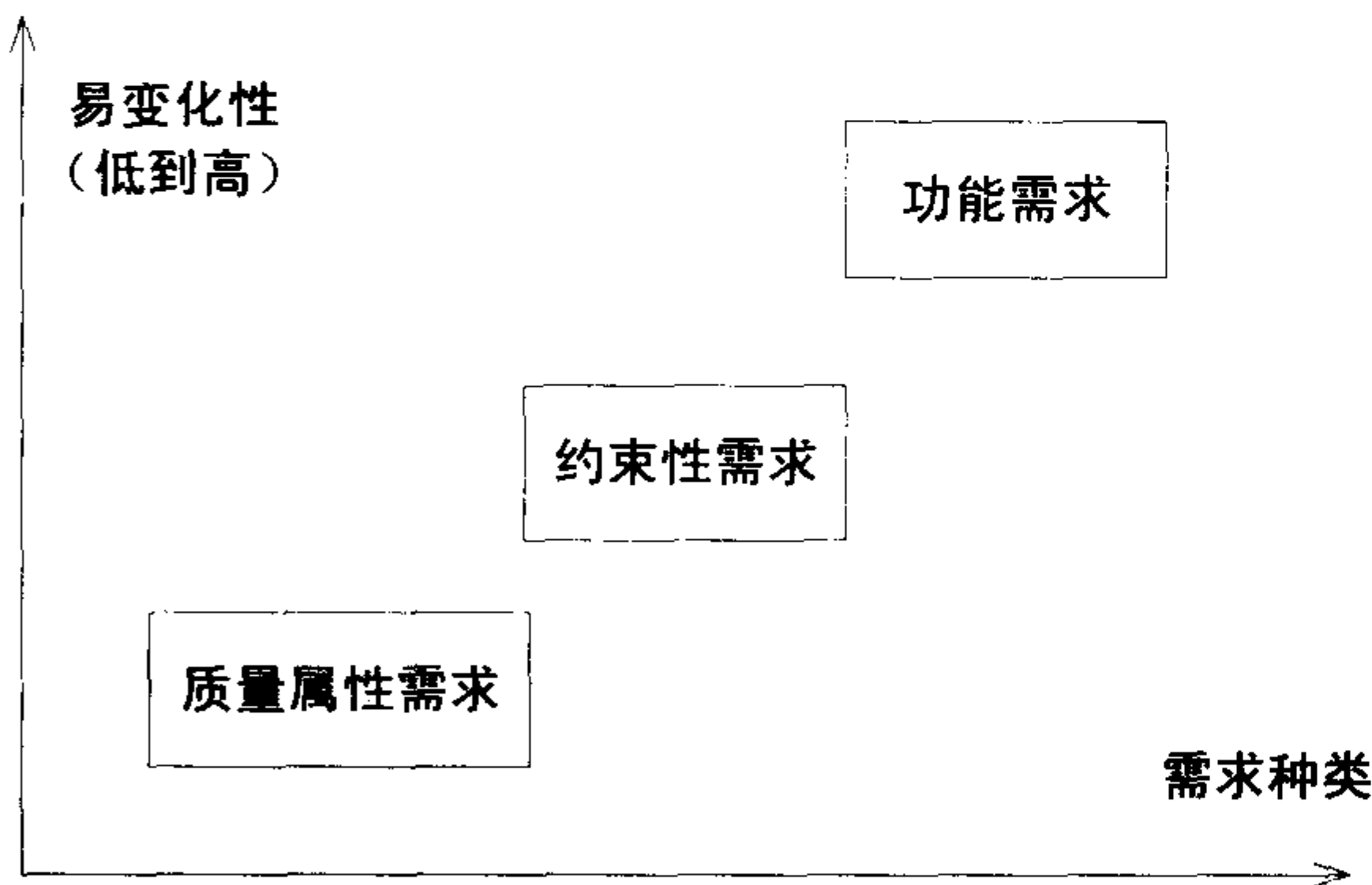


图 10-10 软件需求的易变化性不同

功能需求最容易变化。但同时，任何有经验的软件架构师都必须明白，功能需求的易变性中潜藏着两类不易变性，用好这两类不易变性我们的工作将变得从容些。

一是功能需求集中存在少量长期稳定的功能。例如银行系统中的存钱和支取功能，例如票务系统中的订票功能……这些功能作为软件系统的“基本特色”，任凭软件系统如何升级都会被保留。

二是虽然功能的行为步骤常常变化，但功能点本身趋于稳定。当采用用例技术进行需求捕获和需求分析时，用例图往往是相对稳定的（它描述了功能体系），而用例规约则可能频繁变化（它以交互序列的形式描述功能执行的步骤）。图 10-11 所示为银行系统的例子，我们知道原先

个人储蓄没有利息税，利息税开征之后对“结息”用例规约影响很大，但对用例图并无影响。至于如何将这一点真正用于软件实践并获益，在第 11 章 “专题：用例技术及应用”中将有详细讲解。

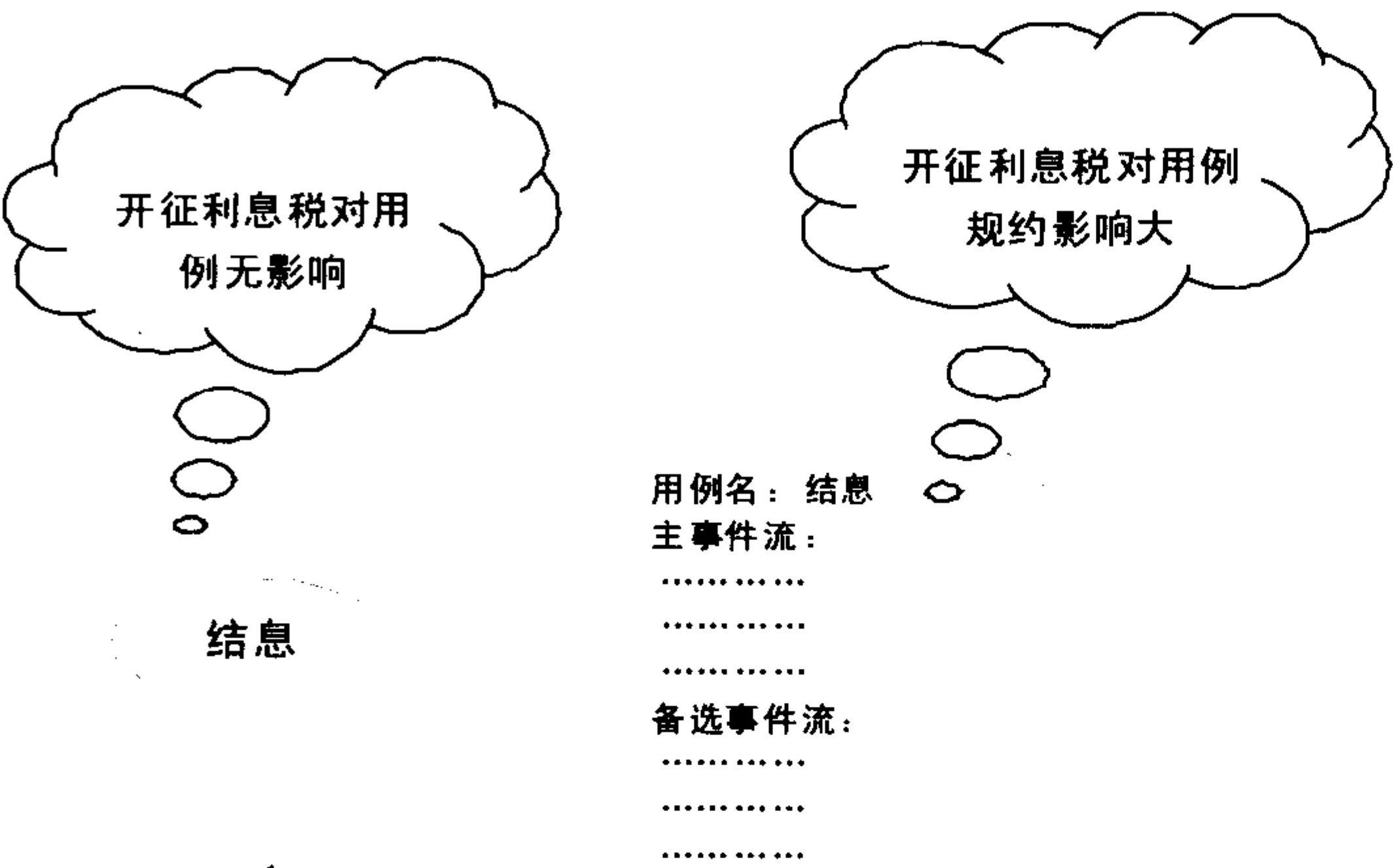


图 10-11 开征利息税对用例和用例规约的不同影响

质量属性需求相对来说最为稳定。例如，但凡是银行系统，总是对安全性要求很高、对持续可用性要求很高等。

一般而言，约束性需求的稳定性则稍差，技术趋势发生变化、法律法规重新界定和用户组织调整改组等，都有可能使约束性需求变更。

10.3.5 质量属性需求与需求折衷

众多质量属性需求之间往往会有冲突，我们必须权衡。在软件实践中，人们对软件质量属性的重视不够，这已成为很多项目遭受失败的常见根源之一。但作为软件架构师，应该对质量属性需求的折衷有系统性的把握。

本章前面比较全面地介绍了 14 种质量属性，而经典文献中给出的是其中 12 种质量属性之间的关系，如表 10-4 所示，，值得架构师研究和借鉴。

由表中可以看出，不同的质量属性之间可能存在 3 种关系：

- 无影响
- 促进
- 阻碍

上表略显混乱，基本无规律可循。本节运用数学中的“矩阵变换”将之变换成更有规律的矩阵表示，如表 10-5 所示。

表 10-4 质量属性关系矩阵（参考《软件需求》）

	持续可用性	性能	可扩展性	安全性	可互操作性	可维护性	可移植性	可靠性	可重用性	鲁棒性	可测试性	易用性
持续可用性								+		+		
性能			-		-	-	-	-		-	-	-
可扩展性		-		-		+	+	+			+	
安全性		-			-				-		-	-
可互操作性		-	+	-			+					
可维护性	+	-	+					+			+	
可移植性		-	+		+	-			+		+	-
可靠性	+	-	+			+				+	+	+
可重用性		-	+	-	+	+	+	-			+	
鲁棒性	+	-						+				+
可测试性	+	-	+			+		+				+
易用性		-								+	-	

说明：“+”代表“行属性”促进“列属性”；“-”代表的含义则相反

表 10-5 调整后的质量属性关系矩阵

	性能	安全性	持续可用性	可互操作性	可靠性	鲁棒性	易用性	可测试性	可重用性	可维护性	可扩展性	可移植性	
性能				-	-	-	-	-		-	-	-	区域 1
安全性	-			-			-	-	-				
持续可用性					+	+							
可互操作性	-	-									+	+	区域 2
可靠性	-		+			+	+	+		+	+		
鲁棒性	-		+		+		+						
易用性	-					+		-					区域 3
可测试性	-		+		+		+			+	+		
可重用性	-	-		+	-			+		+	+	+	
可维护性	-		+		+			+			+		
可扩展性	-	-			+			+		+		+	
可移植性	-			+			-	+	+	-	+		



如前所述，开发期质量属性可以认为是软件项目的内部属性，这些指标的良好深刻影响着开发人员、维护人员、测试人员和开发管理人员的工作。可测试性、可重用性、可维护性、可扩展性和可移植性这些开发期质量属性之间的关系总结如下（如表 10-5 中“区域 3”所示，即不包含开发期质量属性和运行期质量属性之间的关系）：

- 除了不存在影响关系之外，绝大多数运行期质量属性之间存在促进和被促进的关系。例如，可重用性越好，可测试性、可维护性、可扩展性和可移植性就越好；
- 唯一的例外是，可移植性可能对可维护性造成负面影响；
- 很多属性之间是“相互”促进的。例如，可测试性和可维护性是相互促进的关系；
- 但也有不少属性之间的促进关系是“单向”的。例如，可移植性促进可测试性，但可测试性却不能促进可移植性（也不会降低可移植性）。

相比之下，和运行期质量属性相关的制约关系就显得比较复杂。但经过仔细分析，我们还是能把握住其中较为明显的规律：

- 非常典型的是，性能和安全性这两个运行期质量属性和其他所有质量属性之间都是抵触关系（如表 10-5 中区域 1 所示）。
  - 性能和安全性要求高，势必会与其他质量指标带来负面影响。例如，此消彼涨，为了构造更加安全的系统，在系统的易用性、可测试性和可维护性方面就“可能”要做出让步；
  - 反过来，其他质量属性指标也会对性能和安全性造成负面影响。例如，某些嵌入式系统对性能要求非常高，但可测试性和可靠性方面也必须达到极高要求，这时就需要架构师非常高超的折衷技能，以达到总体最优的效果。
- 那么，性能和安全性之间呢？很简单，如表中所示，极高的安全性要求将使我们不得不降低性能要求；
- 除了性能和安全性之外，持续可用性、可互操作性、可靠性、鲁棒性和易用性等运行期质量属性相关的制约关系比较复杂，需要更多经验。

## 10.4 PM Tool 实战：需求分析

### 10.4.1 上游活动：确定项目愿景

一个项目要被开发、要拨款立项，一定有它的业务目标。作为《愿景文档》内容的一部分，业务目标占有非常重要的地位。即使是敏捷软件开发，也很重视以业务目标列表等形式来明确建立某软件系统的最终目的。

从组织的角度而言，之所以要开发并使用 PM Tool 作为项目管理的工具，是要达到以下业务目标（如表 10-6 所示）：通过帮助项目经理更好地控制项目、更有效地分配资源，帮助项目成员之间更顺畅地协作，从而提高项目的投入产出比。

表 10-6 要开发的 PM Tool 的业务目标列表

业务目标列表
<ul style="list-style-type: none"><li>• 帮助项目经理更好地控制项目；</li><li>• 帮助项目经理更有效地分配资源；</li><li>• 帮助项目成员之间更顺畅地协作。</li></ul>

### 10.4.2 第 1 步：从业务目标到特性列表

业务目标是组织或客户方的高层对未来系统的期望，最终要落实到使用这套系统的人（最终用户）实际操作中所需的功能——这些功能被称为“用户需求”。如果从业务目标向用户需求直接过渡的话，我们会发现中间的“跨度”过大，所以可以借助特性列表技术作为中间的“跳板”。

特性（Feature）在实践中其实有两种用法：

- 有时，我们把它看作高度概括的功能性需求，它的数量将比具体的用户需求的数量要少一个数量级。它的作用是支持从业务目标到用户需求的过渡思维：“为了达到期望的业务目标，未来系统应在大方向上具有哪些方面的特性，每个特性再有一组功能来支撑……”
- 另外一些实践者把特性当作比“功能”更小的需求单位，特性驱动开发（Feature Driven Development, FDD）方法也持这种观点；

无论如何，本书认为特性的第一种用法对实践非常有帮助。图 10-12 展示了 PM Tool 案例的从业务目标到特性列表的思维过程。例如，为了帮助项目经理更有效地控制资源，PM Tool 不仅应该支持查看资源的“静态”信息（如职位、技能、特长），还应该支持查看资源的“动态”信息（如当前所承担的工作量）。再例如，为了使项目成员之间更顺畅地协作，应使任务的分配对所有项目成员透明并支持任务的重新分配。

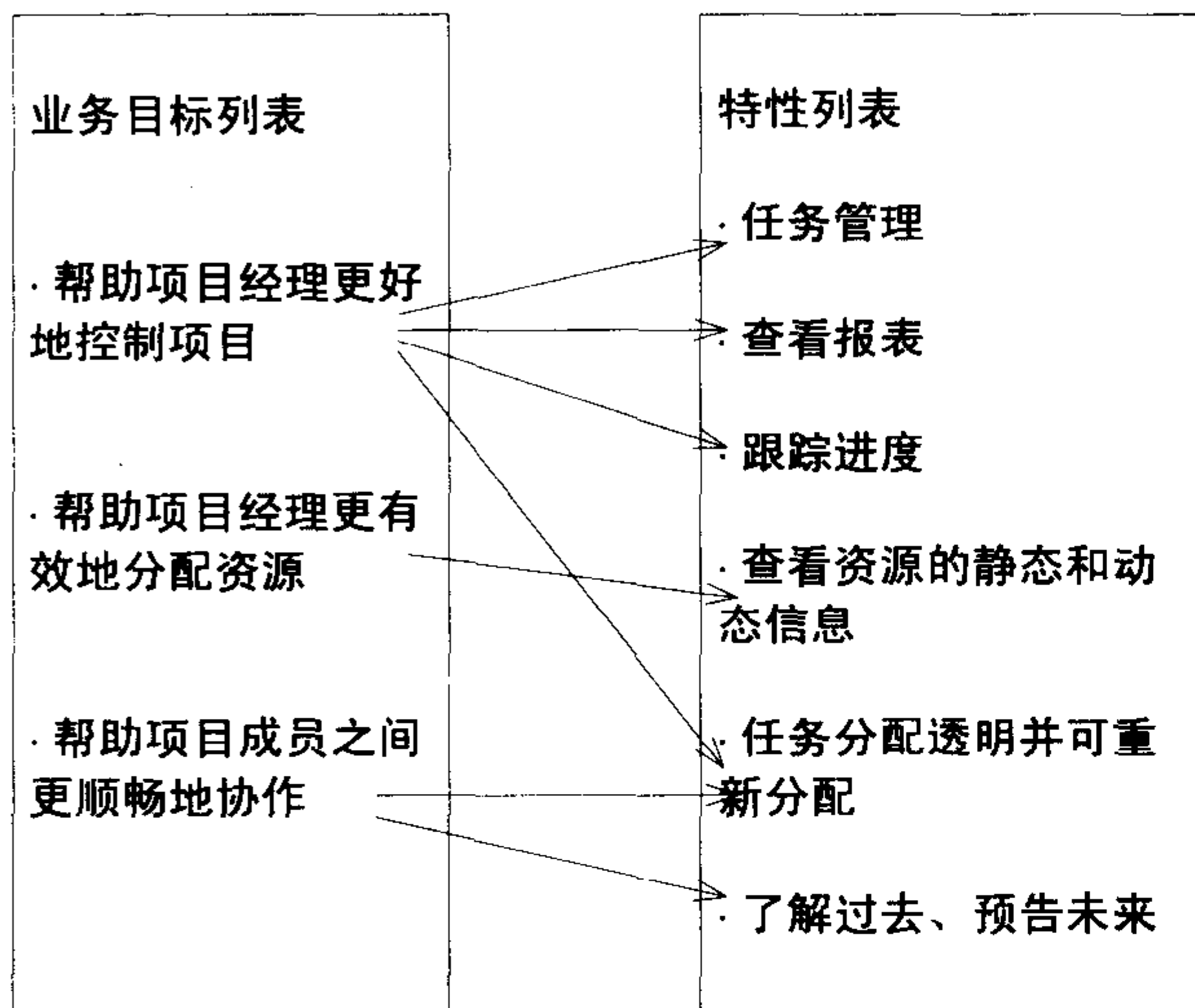


图 10-12 从业务目标到特性列表

### 10.4.3 第 2 步：从特性列表到用例图

所谓用户需求，就是用户希望软件系统能为他做什么。用例图技术是捕获和记录用户需求的合适技术，它是以用户为中心的，用例名是用户需求最直观、最简便的反映。图 10-13 展示了我们根据特性列表的启发和与用户进行讨论所得到的结果：PM Tool 总的用例图。



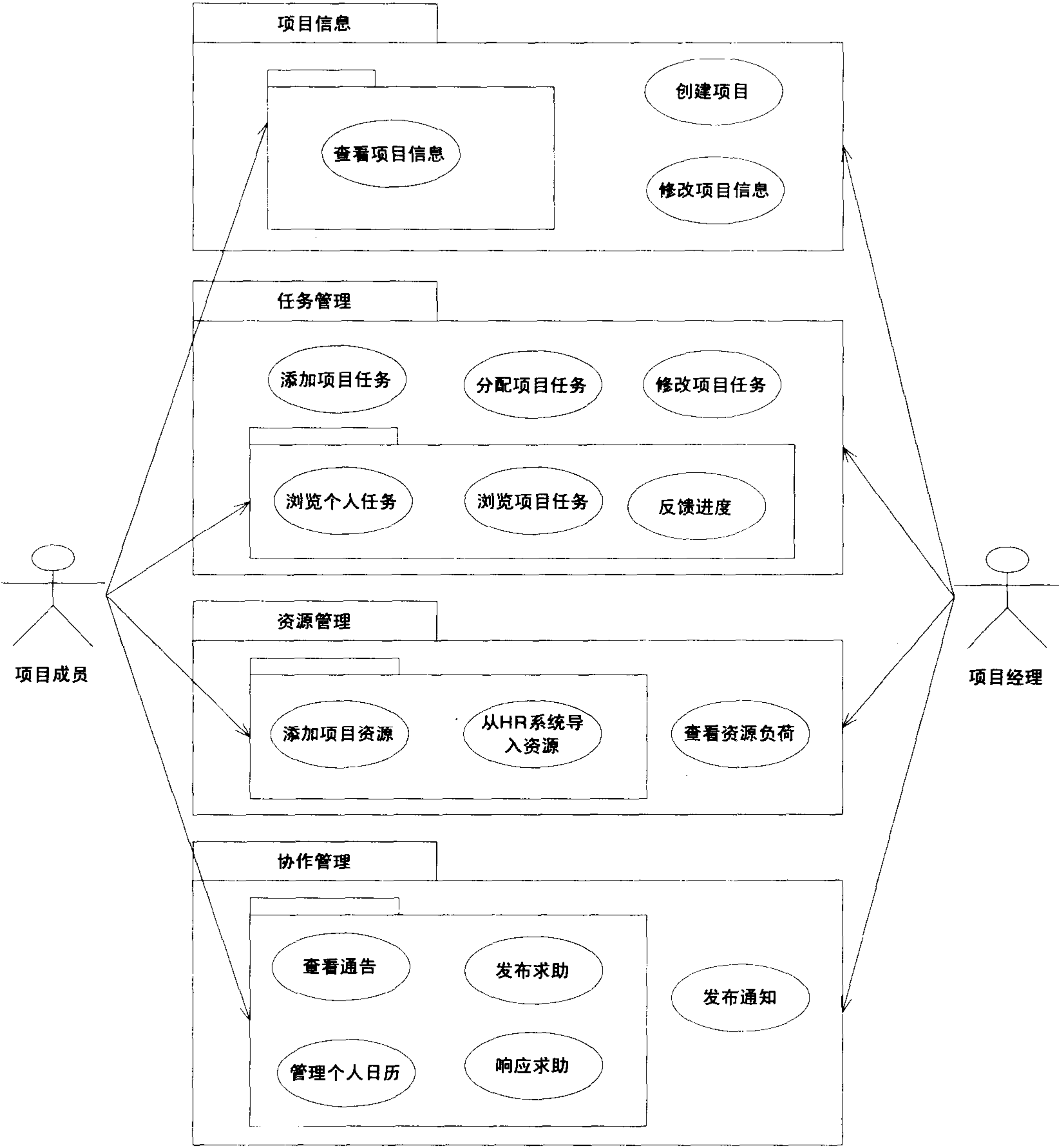


图 10-13 用例图表示的用户需求

从用例图中很清楚地看到：PM Tool 的主要用户分为两种角色，项目成员和项目经理。有些功能是只有项目经理才能使用的，在用例图中通过“包”的方式给予清晰说明。当然，用例图辅以用例简述技术对每个用例的功能进行简要说明效果很好，在此未列出。

10.4.4 第 3 步：从用例图到用例规约

用例图并不足够。换句话说，需求分析进行到用户需求的层次并不足够，因为如果不明确定义软件系统的行为需求，我们就不知道要开发什么。基于用例技术的需求实践中，此时应和用户一起编写用例规约（对于简单并不易产生歧义的需求可以仅通过“用例简述”说明）。

仅举一例，表 10-7 所示为“添加项目任务”的用例规约。

表 10-7 用例：添加项目任务

<div>1. 用例名称: 添加项目任务</div> <div>2. 简要说明: 通过此功能，项目经理可以为项目添加任务。</div> <div>3. 事件流:<div>3.1 基本事件流<div>1) 项目经理进入“添加项目任务”界面。</div><div>2) 系统自动显示已经存在的“项目列表”。</div><div>3) 项目经理选定具体项目，并输入任务的名称、任务目标、开始日期、结束日期等基本信息。</div><div>4) 项目经理确定创建此任务。</div><div>5) 系统完成项目任务的创建。</div></div><div>3.2 扩展事件流<div>如果该任务的开展必须在特定任务完成之后进行，则项目经理可选择一个或多个任务作为“前置任务”。</div></div></div> <div>4. 非功能需求: 操作必须方便直观。</div> <div>5. 前置条件 身份验证：登录用户必须是项目经理身份。</div> <div>6. 后置条件 任务被成功添加到项目，或因任务所在项目还未被创建而退出。</div> <div>7. 扩展点 无。</div> <div>8. 优先级 高。</div>
--

### 10.4.5 需求启发与需求验证

在需求分析过程中需要不断地和客户进行交流，这时客户非常希望能够看到给他带来实感的界面草图甚至可执行的系统原型。而开发方最担心的问题是客户需求的不断变化，所以他们也希望能够尽早掌握客户的真正需求，并希望需求成果得到客户的确认。

为此，我们可以在需求分析期间就开始界面设计（如图 10-14 所示），并将界面草图等设计成果用于和客户的交流当中，这样可以增加实感、方便交流，并帮助客户“发现”他真正想要的功能。当然，界面设计的成果并不应该放入《需求文档》，因为它们是设计而不是需求。

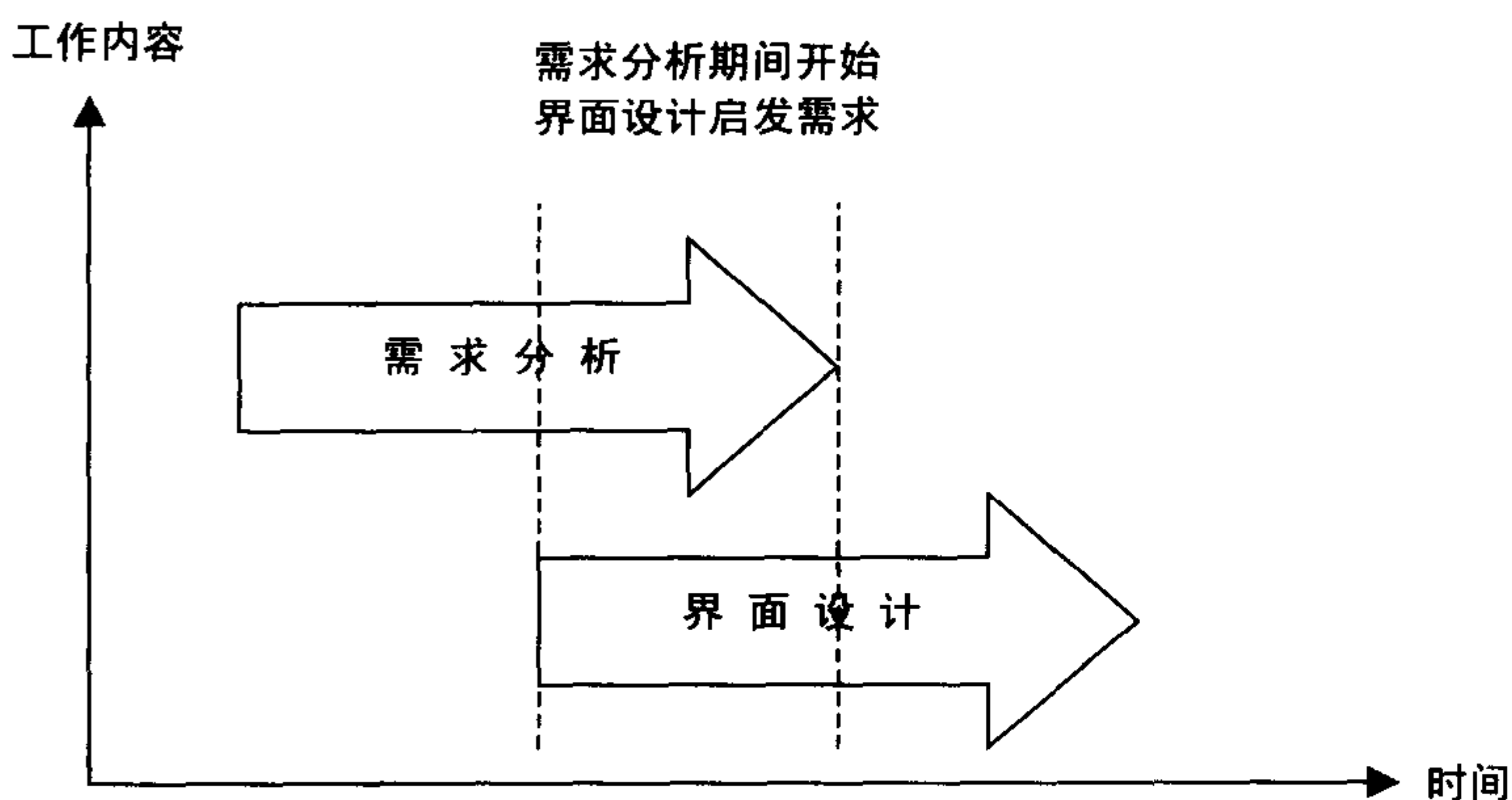



图 10-14 界面设计与需求分析

举个例子。在一开始和客户讨论“添加项目任务”这项需求时，他们未必能意识到两个需求细节：

- 他们希望能为每项任务指定优先级，这样有利于承担多项任务的项目成员在时间不够时权衡取舍；
- 出于对易用性的极高要求，他们要求确定任务细节的时机必须非常灵活。

后来，在界面图和可执行原型的帮助下，用户很容易地意识到了“令人不满的地方到底在哪儿”，明确提出了上面的“两个需求细节”，开发方也更新了“添加项目任务”的界面原型。图 10-15 展示了更新后的界面原型，用户看到了“他想要的 PM Tool”：他既可以只输入任务名称和所属项目就创建任务，也可以同时指定许多详细信息。



 添加项目任务

任务名称 输入文本

添加

所属项目 项目111

取消

可选信息

前置任务

时间安排

所需资源

任务类别 开发任务

父任务 某某任务

进度 0 %

优先级 ☐高 ☒中 ☐低

是否里程碑 ☐里程碑

任务描述 对任务进行详细描述

图 10-15 利用原型启发需求和确认需求

### 10.4.6 最终成果：《软件需求规格说明书》

需求分析工作的最终成果是《软件需求规格说明书》，本书的“11.6 用例与《软件需求规格说明书》”一节专门说明了采用用例建模技术时的《软件需求规格说明书》怎么写，请参考。

非功能需求的满足程度对软件项目的成功非常关键，因此《软件需求规格说明书》必须系统地描述非功能需求的具体要求。非功能需求大致分为质量属性和约束两大类。质量属性是软件系统的整体质量品质——所谓整体品质，就是它往往和大多数功能都有关，而不是仅仅表现在某个功能“内部”。至于约束性需求，它们是架构设计中必须遵循的限制，并有可能“衍生”出质量属性需求和功能需求（可参考“10.3.2 各类需求对架构设计的影响”）。

一部分非功能需求来自用户。诸如性能、易用性等软件质量属性，虽然不像功能需求那样直接帮助用户达到特定目标，但并不意味着软件质量属性不是必需的——恰恰相反，质量属性差的软件系统大多都不会成功。还有一部分非功能需求来自开发者和升级维护人员。软件的可扩展性、可重用性、可移植性、易理解性和易测试性等非功能需求，都属于“软件开发期质量属性”

之列，它们都将影响开发和维护成本。也有一部分非功能需求来自客户组织。架构师必须充分考虑客户对上线时间的要求、预算限制，以及集成需要等非功能需求，还要特别关注客户所在领域的业务规则和业务限制。

表 10-8 归纳了 PM Tool 必须满足的非功能需求。

表 10-8 非功能需求

非功能需求			功能需求
约束	运行期质量属性	开发期质量属性	
客户群工作的平台多样化 成本效益考虑 (PM 的管理成本必须远小于其带来的效益) 应考虑外包趋势 和其他系统交互数据 .....	易用性 (操作方便直观) 互操作性 (可能与 HR 系统、财务系统、Bug 跟踪系统、软件配置管理系统、合作开发单位的 PM 系统等互操作) 跨平台运行 (考虑不同 OS 和 DBMS) .....	可扩展性  .....	参见: · 用例图 · 用例简述 · 用例规约

例如，如果 PM Tool 很难使用，那反而增加了项目组的工作负担，不利于提高效率，因此 PM Tool 必须有较高的易用性。再例如，PM Tool 对可扩展性有一定要求，这也是非功能需求。

## 10.5 总结与强调

本立道生。软件需求方面的知识和能力可谓软件架构师的“起跑线”，缺了“这一块”就意味着输在了起跑线上。

软件架构师面对纷繁复杂的需求，必须对需求分类、需求折衷和需求变更的知识和规律有透彻的了解，从而把握大局、抓住重点，做出最合适的架构设计决策。





## 第 11 章 专题：用例技术及应用

---

实践是最好的老师。

——玛利亚·蒙台梭利

用例仅提供了设计所需要的所有黑盒行为需求。

——Alistair Cockburn, 《编写有效用例》

需求说明书经常长达几十甚至几百页，它一直是开发软件的一个困难来源。主要的困难在于结构方面……

——Ivar Jacobson, 《统一软件开发过程之路》

用例，又是用例。

相信读者会有如下同感：我们掌握的知识本身和我们的实践能力并不成正比。为什么这么说呢？因为如果不能根据经验使“头脑中的知识”和“实践”真正“匹配”起来，知识就无法转化成真正的实践能力。

对实践者而言，仅讨论“用例是什么，用例不是什么”是绝对不够的，因为在实践中，我们其实在不同的时候分别使用和用例相关的不同技术。因此，以用例技术的应用为导向，将不同的技术融入到具体实践场景中去讨论才更有意义。

### 11.1 用例图 vs. 用例简述 vs. 用例规约 vs. 用例实现

---

用例相关的技术比较多，本节将结合银行的储蓄系统案例说明这些技术的不同：

- 用例图
- 用例简述
- 用例规约
- 用例实现

用例图描述软件系统为用户或外部系统提供的服务。如图 11-1 所示，用例图最重要的元素是参与者（Actor）和用例（Use Case）。参与者是与系统交互的角色或系统；参与者既可以是系统的用户，也可以是和系统有直接交互关系的系统。用例描述了系统能为外部参与者提供的功能；用例的名称应该从参与者的角度进行描述，并以动词开头，这样一来通过“读图”可以清晰地获得用例图的语义，例如图 11-1 的储蓄系统中，通过“读图”得知“柜员开户”和“柜员销户”等功能。

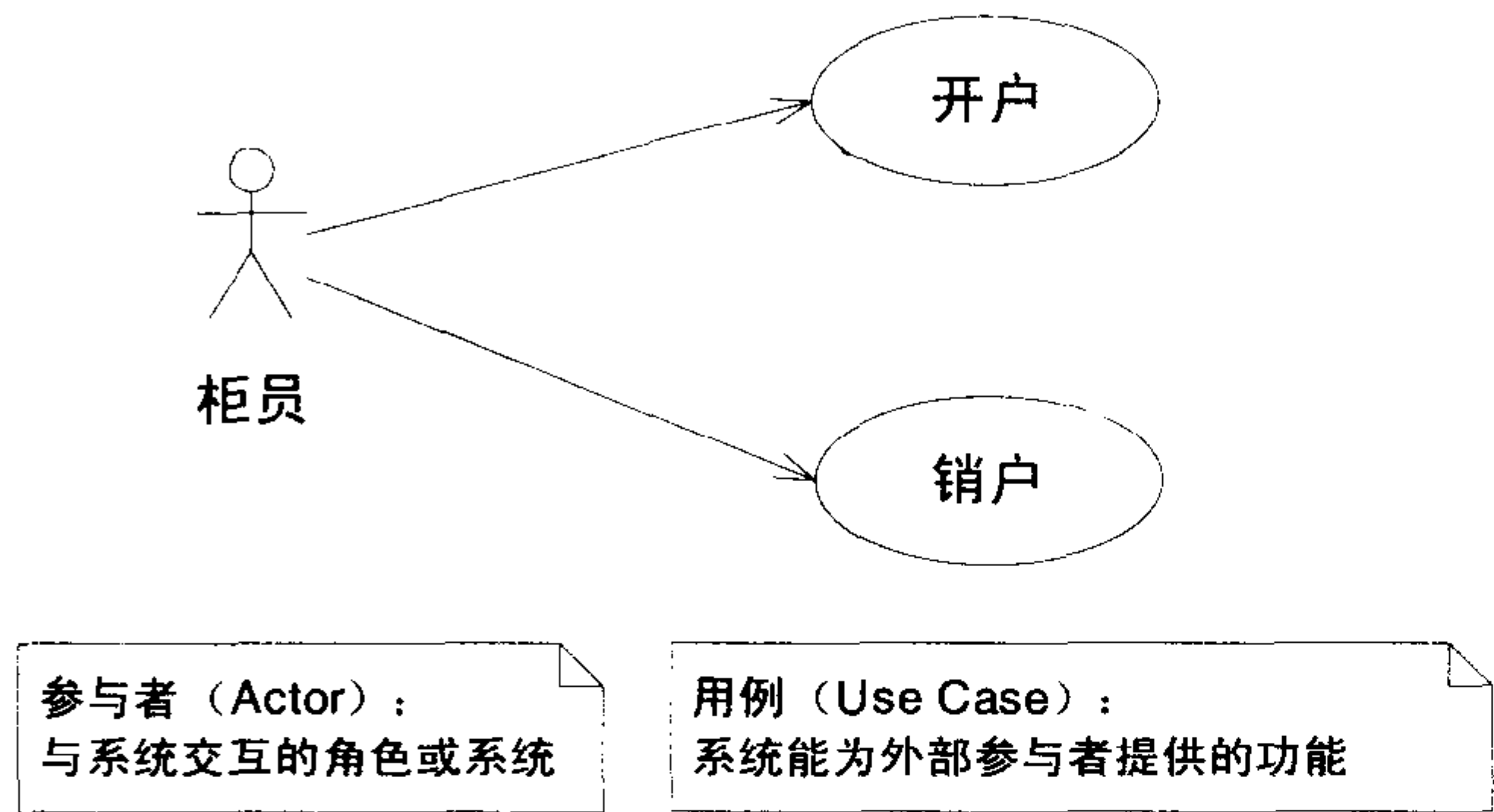


图 11-1 储蓄系统用例图的一部分

用例图通过确定与本系统交互的角色或外部系统、描述系统必须提供的功能的方式，清晰地界定了系统的功能范围（Scope）。用例图不仅是可视化的，而且是结构良好的、有利于从宏观上反映系统功能的大局观。

用例图所含的用例必须被命名，但用例图所含的用例信息也仅此而已了。因此，还需要其他的“用例说明技术”对用例进行更进一步说明。用例简述就是“用例说明技术”中最简单、最实用的一种。所谓用例简述，就是通过简短的文字对用例的功能进行描述；一般而言，用例简述都应包含成功场景的简单描述。如表 11-1 所示，它对储蓄系统“销户”用例进行了简要描述。

表 11-1 储蓄系统“销户”用例简述

用例名称：销户
用例简述：帮助银行工作人员完成银行客户申请的活期账户销户工作。需客户提供证件和密码。
优先级：高

用例简述是一项非常轻便的技术，很多敏捷方法都通过类似用例简述的技术捕获和交流需求，例如极限编程（Extreme programming，XP）的用户故事卡，以及特征驱动开发（Feature

Driven Development, FDD) 的特征等。并不是所有功能需求都是复杂的，因此如果用例简述已足够明确，那么就不必再对用例进行更为细化的说明了，节省了时间和资源。本书建议，为了对软件系统所提供的功能进行泛而不深的总体描述，可以采用“用例图+用例简述”的方式，详见“11.4 用例与需求捕获”。

再说说另一种用例描述技术——用例规约。用例规约是对用例的详细描述，一般包括简要说明、主事件流、备选事件流、前置条件、后置条件和优先级等。用例规约的主要目的是界定软件系统的行为需求（需求可以划分为业务需求、用户需求和行为需求三个层次，所谓行为需求是指软件系统为了提供用户所需的功能而必须执行哪些行为）。表 11-2 展示了储蓄系统“销户”的用例规约。

表 11-2 销户用例规约

1.	用例名称: 销户
2.	简要说明: 帮助银行工作人员完成银行客户申请的活期账户销户工作
3.	事件流: 3.1 基本事件流 1) 银行工作人员进入“活期账户销户”程序界面; 2) 银行工作人员用磁条读取设备刷取活期存折磁条信息; 3) 系统自动显示此活期账户的客户资料信息和账户信息; 4) 银行工作人员核对销户申请人的证件, 并确认销户; 5) 系统提示客户输入取款密码; 6) 客户使用密码输入器, 输入取款密码; 7) 系统校验密码无误后, 计算利息, 扣除利息税(调用结息用例), 计算最终销户金额, 并打印销户和结息清单; 8) 系统记录销户流水及其分户账信息。 3.2 扩展事件流 ① 如果存折磁条信息无法读出, 需要手工输入账号。 ② 如果销户申请人的证件与客户资料信息不符或其他业务因素, 而不予受理的, 银行工作人员直接退出。 ③ 如果系统密码校验错误, 提示重新输入密码; 密码校验失败超过 3 次, 系统提示并自动退出。
4.	非功能需求: 申请受理处理的过程操作事件应在 30 秒内。 打印的销户和结息清单应该清晰明了。
5.	前置条件



	账户为正常状态（即不是挂失、冻结或销户状态）。
6.	后置条件 销户成功并将销户信息存入数据库，证件不符而退出，密码不符而退出。
7.	扩展点 无
8.	优先级 高

用例简述和用例规约都是对用例进行说明的技术，但用例规约要比用例简述复杂 10 倍以上，那么为什么还需要用例规约技术呢？在应用用例规约技术时应注意哪些问题呢？现归纳如下：

- 用例规约中对系统行为的描述是以用户为中心展开的，便于和用户交流；
- 用例规约既关注主事件流所描述的成功场景，也关注备选事件流所描述的异常场景，有利于促进系统化思维、发现异常场景、完善系统功能和提高易用性；
- 很多项目运用用例规约技术时采取“一刀切”的态度，这是不对的。这个问题很普遍。他们将所有用例统统细化到用例规约程度，往往是所有程序员都被发动起来突击写用例规约，这种“为写而写”的做法浪费了时间和资源，但意义却不大，正确的做法请参见“11.5 用例与需求分析”。根据笔者的经验，这个问题产生的根源和编写《软件需求规格说明书》有关，故在 11.6 节中会对“用例与《软件需求规格说明书》”进行专门讲解；
- 在实践中我们可以对用例规约的格式进行剪裁甚至扩充，以满足具体工作的需要。例如，可以增加用例的“使用频率”，以供界面设计和性能设计时进行参考。例如，可以增加“需求背景及可能的变化”，供架构师设计可扩展的软件架构时进行参考；
- 用例规约中应该包含用户界面原型吗？答案是不应该。界面如何规划本质上是属于设计的工作，将一部分设计混入需求而确定下了，会造成后续设计工作的被动。但实践中推荐在需求分析时就开始用户界面的设计（以及开发水平抛弃原型），以便对需求交流起到辅助作用；
- 后置条件应覆盖所有可能的用例结束后的状态。即，后置条件不仅仅是用例成功结束后的状态，还应该包含用例因发生错误而结束后的状态。例如表 11-2 所示的储蓄系统“销户”用例中，“证件不符而退出”和“密码不符而退出”就是因错误而结束的情况。

那么，何为用例实现呢？在面向对象的理论体系中，协作被定义为“多个对象为了完成某种目标而进行的交互”。而用例实现是协作的具体运用：即为了实现用例定义的功能，我们必须考虑需要哪些类，这些类又如何交互来完成最终的功能。图 11-2 展示了储蓄系统销户的用例实现。

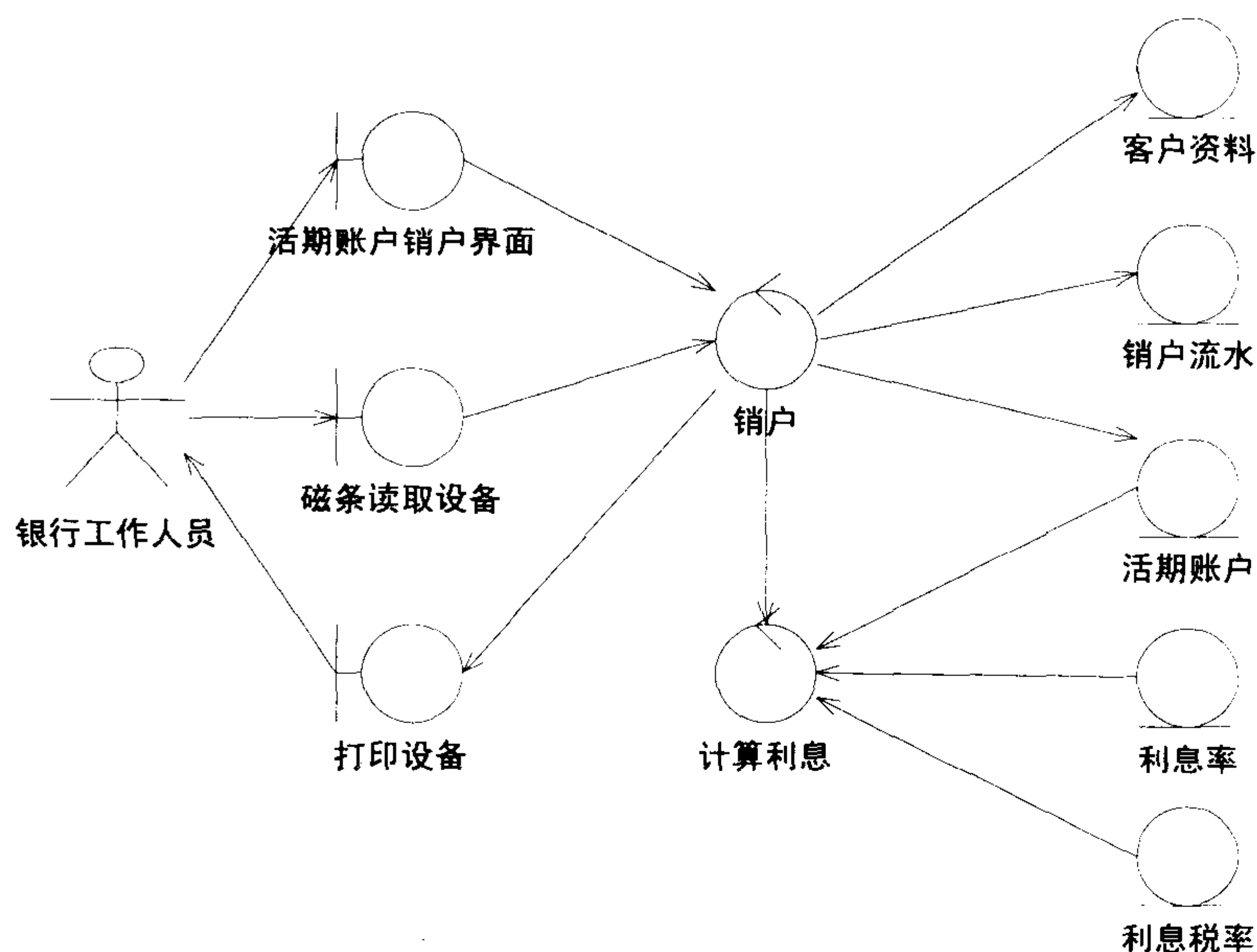


图 11-2 销户的用例实现

不难看出，用例实现是从功能需求向设计方案过渡的纽带。而且，通过分析一组关键用例的用例实现，可以获得未来系统的理想化的职责模型，它的重点是识别组成软件系统的高级职责块、规划它们之间的关系；这个职责模型是规划架构机制、满足高质量属性要求的武器。

研究用例相关技术和需求的关系对实践很有帮助（比如用例到底描述用户需求还是行为需求），所以下面将结合软件需求的层次理论来总结一下用例技术。

要开发一个软件系统，不同层次的涉众提出需求所站的立场不同，因此就产生了需求的层次。用户方的决策层会站在组织高度、从业务层面提出需求；用户更关心自己的工作如何完成、需要哪些功能来辅助；而最后要开发的，是实实在在的软件要提供的行为。这就是需求的三个层次：

- 业务需求：组织要达到的目标；
- 用户需求：用户使用系统来做什么；
- 行为需求：开发人员需要实现什么。

那么，一个很现实的问题：用例和需求层次的关系是什么？虽然有一些书中陈述了自己的看法，但用例到底描述的是用户需求还是行为需求，都不能给出令人信服的解释。笔者认为，用例

技术其实是多种技术组成的“技术族”，不应该一概而论，而是必须具体到某个技术谈其应用。具体而言，用例图从总体上反映了用户需求，而用例简述和用例规约分别是行为需求的简化描述和详细描述（如图 11-3 所示）。至于用例实现，已属于设计范畴了。

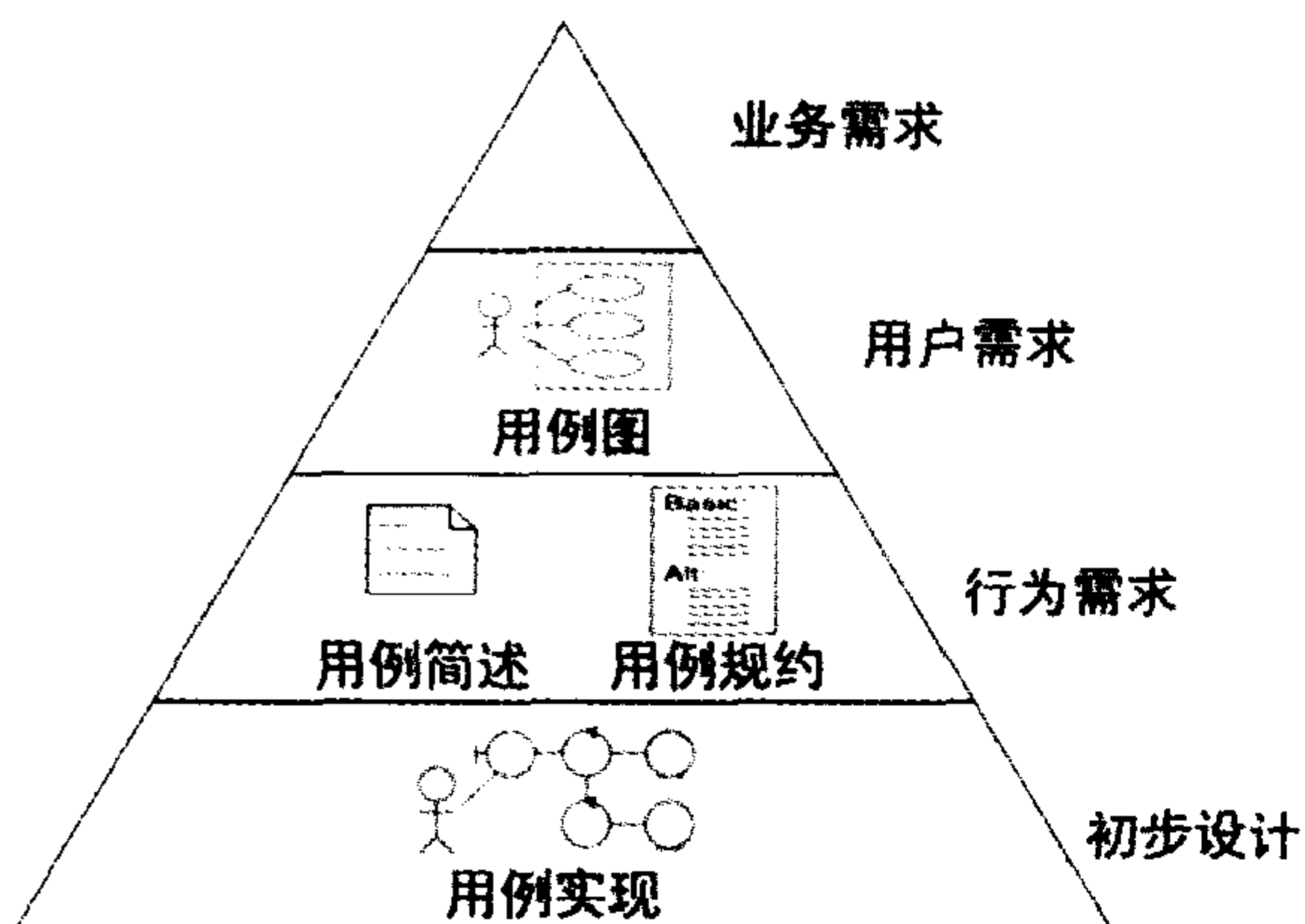


图 11-3 几种用例技术和需求的关系

概括而言，基于用例技术开展需求活动有以下好处：

- 用例方法是以客户为中心的。它完全站在用户的角度上，从系统的外部来描述系统的功能，每个用例代表一个完整的系统服务；
- 用例方法比传统的 SRS 更易于被用户所理解，是开发人员和用户之间进行沟通的有效手段之一；
- 用例图可以从大局上反映系统的功能结构，并且用例图在很大程度上不会受到需求变更的冲击——因为它不涉及需求细节；
- 传统需求方法采用功能分解方式，非常容易混淆需求和设计的界限，而用例方法利于解决这个问题。

## 11.2 储蓄系统案例：需求变化对用例的影响

很多人都认同，需求变化是软件开发中“最令人痛苦的事”。研究需求变化造成影响的规律，从而更有效地应对需求变化，是极有意义的。下面结合银行的储蓄系统案例，说明“开征利息税”这一需求变化对用例的影响。

上一节中介绍的储蓄系统一直工作得很好……到了 1999 年 11 月 1 日，我国就开始对个人储蓄存款利息征收个人所得税。这就势必要求我们对原有储蓄系统进行升级，在结息功能、票据格式和数据库结构等方面支持利息税特性。



首先来回顾储蓄系统的功能（实际工作中也是如此）。图 11-4 展示了储蓄系统用例图的一部分，包含开户、销户、结息这 3 个用例。柜员可以执行开户和销户功能；而结息是在每年的结息日（6 月 30 日）例行执行的，同时，销户时也会引起结清利息的处理。

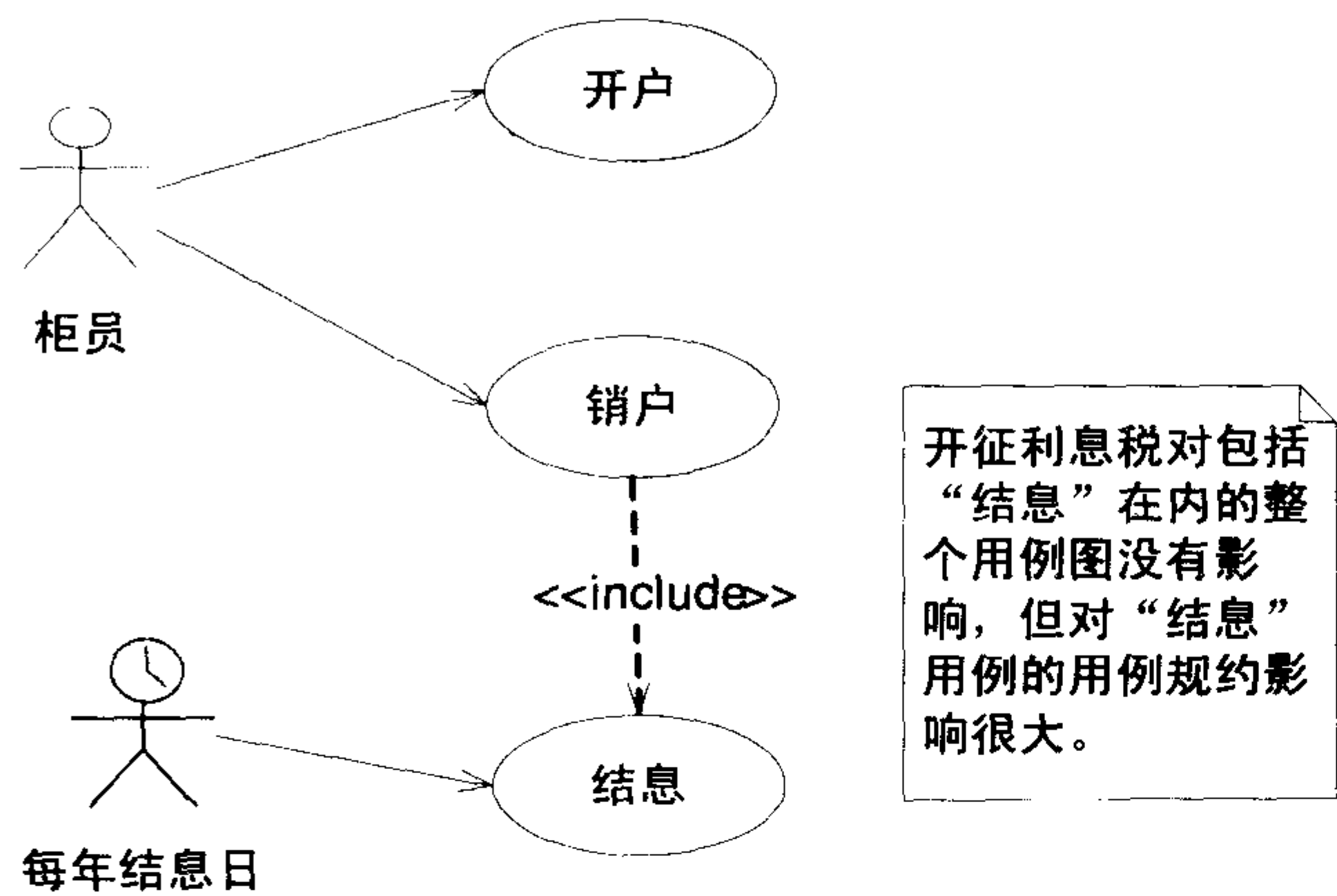


图 11-4 含结息用例的用例图

通过分析，我们发现“开征利息税”对用例图没有影响。这是因为，用例图描述了软件系统的整体功能结构，其中对用例本身的描述一般仅包括用例名。可以肯定“开征利息税”将影响结息功能，但用例图中仅反映了“储蓄系统必须有结息功能”，所以并不受“结息业务规则变化”的影响。

进一步看，开征利息税将影响的是“结息”用例的用例规约。如表 11-3 所示，主事件流必须增加扣除利息税的相关步骤，其他步骤也有重要改变。至于另一种用例描述技术——用例简述，一般也不会受到影响，因为它并不包含处理步骤的详细描述。

表 11-3 结息的用例规约

1.	用例名称: 结息
2.	简要说明: 完成银行活期账户结息的工作.
3.	事件流: 3.1 基本事件流 1) 系统发出结息命令; 2) 系统查询此账户的开户日期、存款利率、存款积数、利息税率等信息; <b>//此步改变</b> 3) 系统计算此账户的利息=存款积数×存款利率;

1)	系统发出结息命令;
2)	系统查询此账户的开户日期、存款利率、存款积数、利息税率等信息; <b>//此步改变</b>
3)	系统计算此账户的利息=存款积数 × 存款利率;
4)	系统计算此账户的利息税=利息 × 利息税率; <b>//此步新增</b>
5)	系统计算出应付利息=利息 - 利息税。 <b>//此步新增</b>
6)	系统将结息信息记账。 <b>//此步改变</b>
3.2 扩展事件流	
无	
1.	非功能需求: 执行国家规定的当前利息率、利息税率。 <b>//此条改变</b>
2.	前置条件 在规定结息日批量结息, 或由销户交易触发。
3.	后置条件 将结息信息存入数据库。
4.	扩展点 无
5.	优先级 高

总结一下。通过此案例的分析我们意识到，需求变更并不会造成“席卷一切”的变化，它所造成的影响是有规律的。一个软件系统，它应当提供哪些功能往往是和业务目标相一致的，而一个企业的业务目标还是相当稳定的（虽然业务环境不断变化），因此用例图所反映的功能总体结构发生变更的几率较小。察其究竟，对软件开发影响巨大的需求变更其实更多地发生在行为需求层面——用例规约的主事件流和备选事件流正是反映行为需求。于是，我们惊喜地发现了需求变化波及不同需求工件的规律：

- 用例图作为功能需求的泛而不深的总体描述，它是比较稳定的；
- 用例简述不涉及细节，因此也比较稳定；
- 用例规约通过交互序列的方式描述功能需求，受需求变化影响较大。

### 11.3 用例技术应用指南

如何运用不同的用例相关技术，图 11-5 进行了归纳。

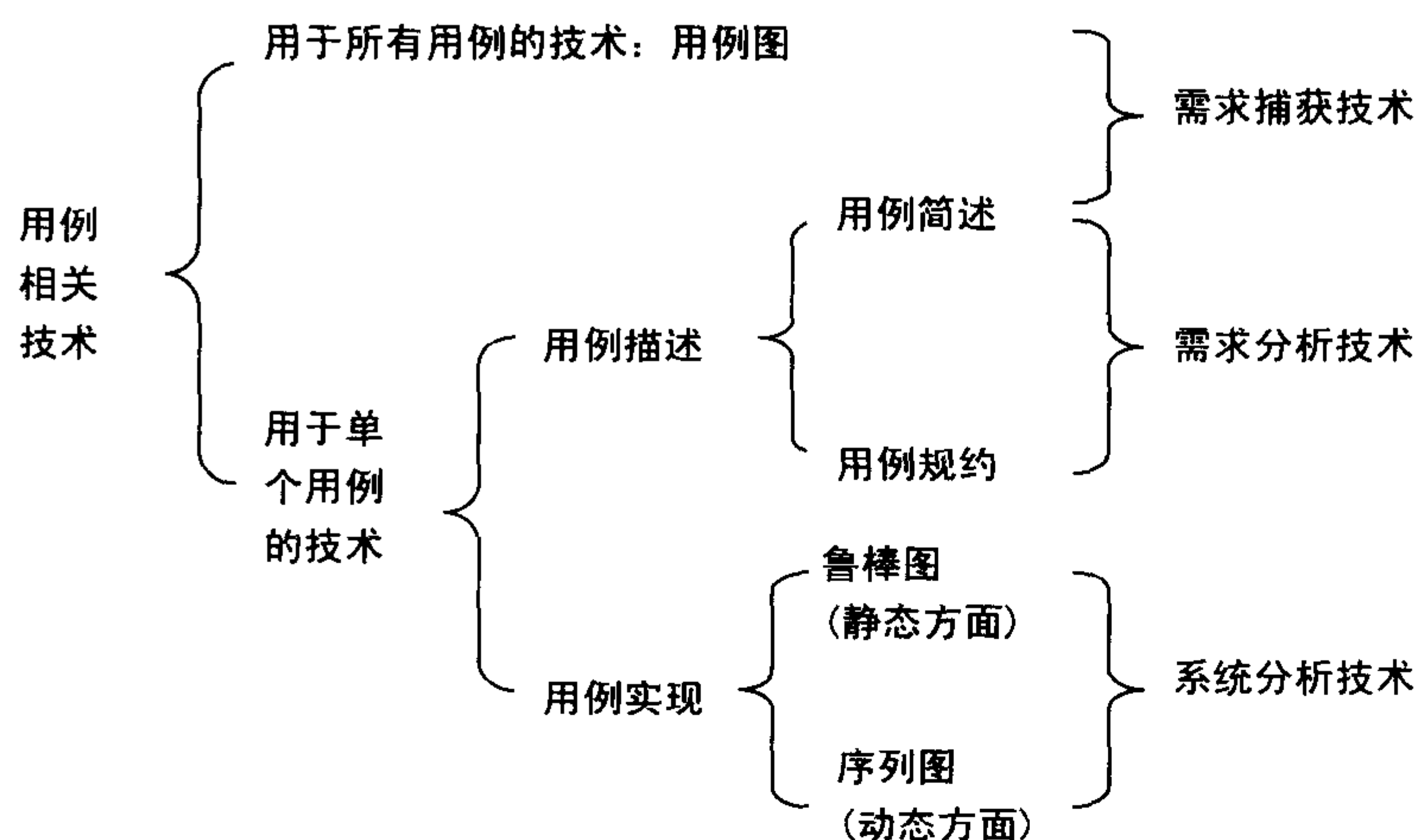


图 11-5 如何运用不同的用例相关技术

由图中可见，从大的方面首先可以将“用例相关技术”分为两类。

**用于所有用例的技术。**这就是用例图，它的好处是形象、清晰和总体观好。另外，由于通过椭圆形代表的用例明确了“什么在系统内”，通过人形代表的参与者明确了“什么在系统外”，从而用例图在界定软件系统的功能范围（Scope）方面表现得非常优秀。

**用于单个用例的技术。**此类技术对单个用例进行行为需求的简化描述和详细描述。

根据各自的目的不同，“用于单个用例的技术”又可以分为两大类：

**用例描述技术。**这类技术都是以描述单个用例为目的的，根据描述的详细程度不同而进一步分为两类：用例简述和用例规约。所谓用例简述，就是通过简短的文字对用例的用户、目标、功能和行为规则等进行描述；一般而言，用例简述都应包含成功场景的简单描述。用例规约是用例的完整而系统的描述，不仅包括了主事件流、备选事件流、前置条件和后置条件等，还可以包括用例 ID、优先级和使用频度等方面的定义。

**用例实现技术。**“用例实现”是通过一组对象相互交互的方式来实现用例所定义的功能的一个协作。

在实践中，用例图和用例简述比较适合用作需求捕获技术。好处有二：一是覆盖面广，所有的功能需求都能被覆盖到；二是深入的细节不多，不易受到需求变更的冲击。

另一方面，用例规约本身是一种思维工具。它围绕“能为特定主要参与者带来价值的可见结果”展开，有助于需求分析人员将各种场景思考清楚，特别是有助于发现所有可能的异常情况，使需求定义更完善。因此，在实践中我们一般将用例规约用作需求分析技术（需求分析的目的是得到明确和规范的需求定义）。



至于用例实现技术，它是一种以建立初步设计为目标系统分析技术（需求分析和系统分析的关系请参见“10.1.2 需求捕获 vs.需求分析 vs.系统分析”）。

## 11.4 用例与需求捕获

需求捕获是知识采集的过程，致力于收集用户对未来软件系统的期望和具体要求。实际的需求捕获工作中，人们可能采用不同的技术（如图 11-6 所示）：

- “需求采集卡”的使用相对广泛。结合客户访谈，将需求的初步信息进行系统化的收集；
- 采用 XP 过程进行敏捷开发的团队，他们会采用“用户故事卡”。故事卡是一张记录着“用户故事”的索引卡，开发人员要实现故事卡上规定的功能的时候，可以随时请现场客户给予更详细的讲解。可见，XP 要求客户作为团队的一份子和具体开发人员一起工作（即“现场客户”），在某种程度上是要代替传统意义上的《需求规格说明书》；
- 当采用用例技术进行功能需求捕获时，可能有不同的做法，采用“用例图+用例简述”是比较轻量的做法。本书推荐这种做法；
- 一开始就采用“用例规约”技术有些过于笨重了。对于没有歧义的简单需求，用例简述就足够了。对于复杂的功能需求，则可以进一步通过用例规约进行分析探索。用例规约本质上属于需求分析技术，因为它偏重于对知识进行系统地挖掘和整理（这是需求分析的任务）；通过需求捕获难以得到用例规约。

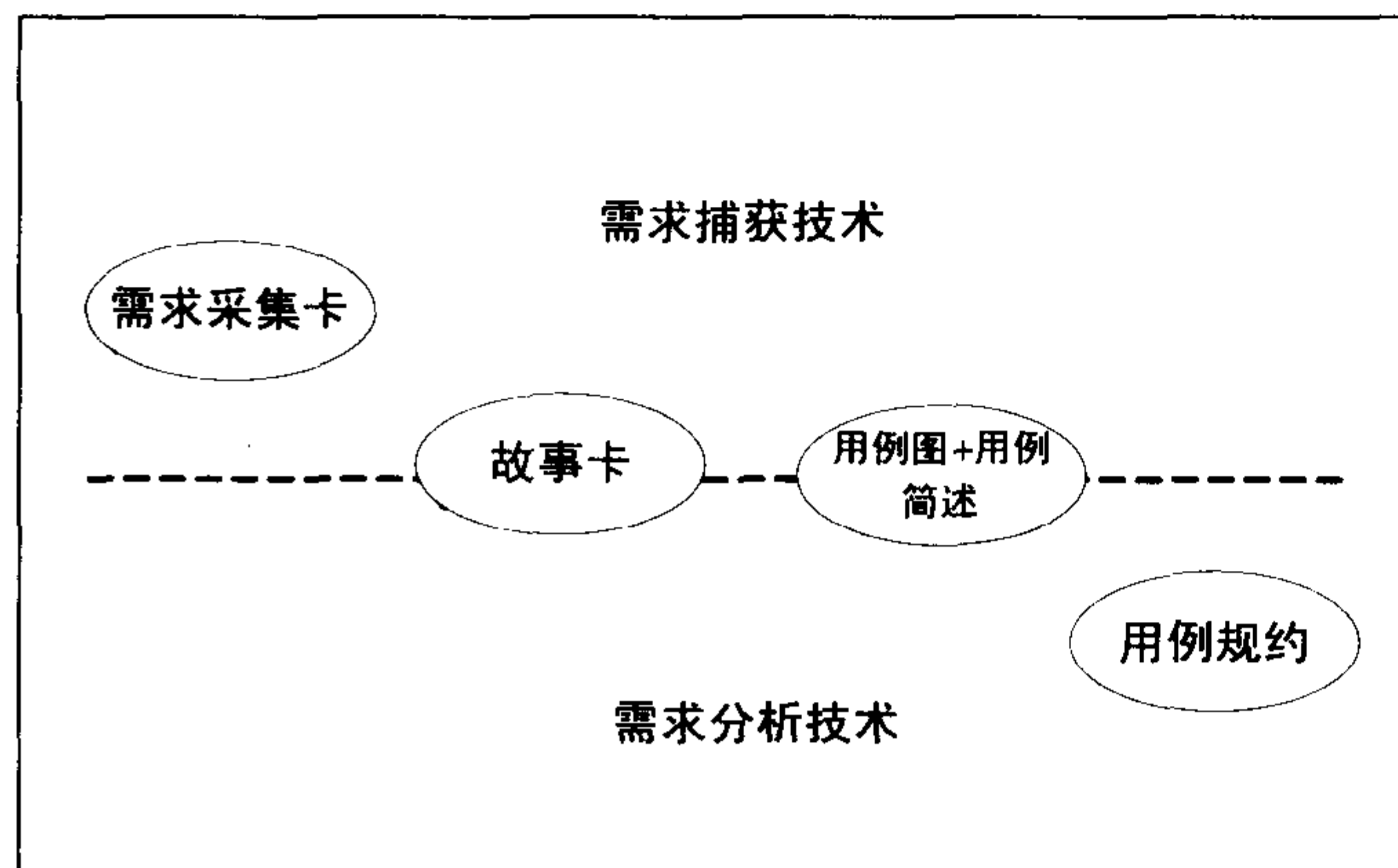


图 11-6 需求捕获的不同技术所处的位置

为什么推荐采用“用例图 + 用例简述”的组合进行需求捕获呢？因为这样做既获得了敏捷的优点，又不失功能需求的总体观（一些敏捷方法的缺点）。用例图能够直观和系统地反映了哪些功能属于系统内部，哪些人、系统和设备属于系统外部并与本系统进行交互。用例简述对所有用例的功能进行简要说明。

## 11.5 用例与需求分析

需求分析的目的在于以比较规范的形式，明确定义软件系统的需求，显然用例规约正是一项规范、明确的技术——它通过特定格式的文本来记录参与者和系统之间的各种情况下的交互场景，以此来明确对软件系统的行为需求。

同时，需求分析还必须去伪存真、查漏补缺；所谓“去伪存真”是指用户表达的不一定是他真正想要的，需求分析员有义务真正了解他们的需求；所谓“查漏补缺”是使需求集完整化、系统化的过程。用例规约技术能帮助需求分析员以用户为中心进行思考，定义不同场景下的软件系统应有的行为需求。

但是，用例规约技术比较“重”，使用不当开销会很大。

首先，架构设计不应等到所有用例被细化到用例规约程度才开始，因为没有必要。对架构设计起关键作用的功能需求只占功能需求的一小部分；这部分用例应该已经被细化到用例规约的程度，它们和其他非功能需求一起决定架构设计方案。

其次，必须聪明地应付需求变更。正如本章前面所述，需求变更对用例图和用例简述的影响比较小，而对用例规约的影响很大。因此我们应该“推后用例细化、激发需求变更”。具体而言：

- 推后用例细化。不是对软件架构起关键作用的用户，可以推迟到要实现该用例所定义的功能之前才进行细化，过早地为这些用例制定用例规约会增加“需求变更管理”的开销，使需求变更的影响增大；
- 激发需求变更。这一点很重要。必须通过不断增加功能、发布小版本、提供给用户试用、接受用户反馈等一系列的活动，让用户一步步明确自己真正想要的功能。对前期版本的反馈中，往往涉及比较多的需求变更，而此时大量用例还没有被细化，所以避免了需求变更造成的巨大冲击。

总之，在项目前期并不将所有用例细化，而是将大部分用例保持在“用例图+简短描述”的水平——这样做并不影响开发出原型来启发和确认用户需求，并可能尽早激发需求变更的发生——直到变更的几率比较小的时候甚至到了程序小组要实现该用例的时候，再由深入到小组的系统分析员对用例进行细化。

## 11.6 用例与《软件需求规格说明书》

将“用例图建模”和“用例规约”技术区别对待有重要的实践意义，例如，我们可以更好地利用《软件需求规格说明书》的格式了。正如 Ivar Jacobson（RUP 的最初缔造者）所指出的：“需求说明书经常长达几十甚至几百页，它一直是开发软件的一个困难来源。主要的困难在于结构方面……”所以我相信 RUP 推荐的《软件需求规格说明书》必定是结构合理的。

1. 前言
  - a) 目的
  - b) 范围
  - c) 定义、缩写词、略语
  - d) 参考资料
2. 需求概述
  - a) 用例模型
  - b) 限制与假设
3. 具体需求
  - a) 用例描述
  - b) 外部接口需求
    - i. 用户接口
    - ii. 硬件接口
    - iii. 软件接口
    - iv. 通信接口
  - c) 质量属性需求
    - i. 性能
    - ii. 易用性
    - iii. 安全性
    - iv. 可维护性……
  - d) 设计和实现约束
    - i. 必须遵循的标准
    - ii. 硬件的限制……



的确如此，使用 RUP 提供的需求文档模板有非常明显的优势：

- 文档的第 2 节和第 3 节分别是“需求概述”和“具体需求”，先概述后详述，增加了文档的层次，而不再是平铺直叙；
- “需求概述”一节的“用例模型”部分归档用例图和用例简述，“具体需求”一节的“用例描述”部分归档重要用例的用例规约；
- 只需阅读页数不多、内容精炼的“需求概述”部分，就可以把握需求的总体信息；
- “需求概述”部分应保证全面性，从而使“具体需求”部分可以只详述部分最关键、最易产生分歧的需求，为开发团队在需求实践方面赢得主动奠定了基础。

## 11.7 总结与强调

---

本章是笔者在经历了从困惑到明朗的实践过程之后的经验总结，希望对读者有用。

如果要用一句话总结，本章的精髓就是：在实践中，我们应该在不同的时候分别使用和用例相关的不同技术。笔者相信，随着用例技术实践的深入，会有越来越多的人认识到将用例相关的不同技术区别对待的好处。



## 第 12 章 领域建模

---

分析的另一种重要产品是领域模型，其目标是使负责该系统基本行为的所有核心类可视。

—— Grady Booch, 《面向对象项目的解决方案》

软件的核心是它为用户解决领域相关问题的能力。

—— Eric Evans, 《领域驱动设计》

模型的选择会影响最终产生的系统的灵活性和可重用性。

—— Martin Fowler, 《分析模式》

《高效能人士的七个习惯》算是一本“名书”了。读过之后，我印象最深的是 4 个字：由内而外。它的第 1 章名为“由内而外全面造就自己”，而最后一章名为“再次由内而外造就自己”，很有意思。

如果给我一次杜撰这句话的机会，我会说：由内而外造就软件。因为我们在软件开发中遇到的很多失败项目，都是由于对问题领域认识不足而引起的；还有许多项目，模型设计不够合理，任何需求的变更或新功能的增加都可能引起一连串的问题。换句话说，忽视了领域模型这个“内”，而仅重视编写程序这个“外”，多半是要出问题的。

这一点不仅和“以客户的需求为中心”不矛盾，它们反而是相辅相成的。例如，Martin Fowler 在《分析模式》一书中就曾经指出：“在对象开发过程中一个很重要的原则就是：要设计软件，使得软件的结构反映问题的结构。……模型的选择会影响最终产生的系统的灵活性和可重用性。”

想让自己的软件系统随需应变吗？请给软件一颗支持变化的“心”。

### 12.1 领域模型基础知识

---

领域建模是建立领域模型的过程。领域建模专注于分析问题领域本身，发掘重要的业务领域概念，并建立业务领域概念之间的关系。接下来，我们来介绍一下什么是领域模型，以及常用于



表达领域模型的 UML 图（从而带来实感）。

12.1.1 什么是领域模型

案例先行。图 12-1 展示了银行领域模型的一小部分。

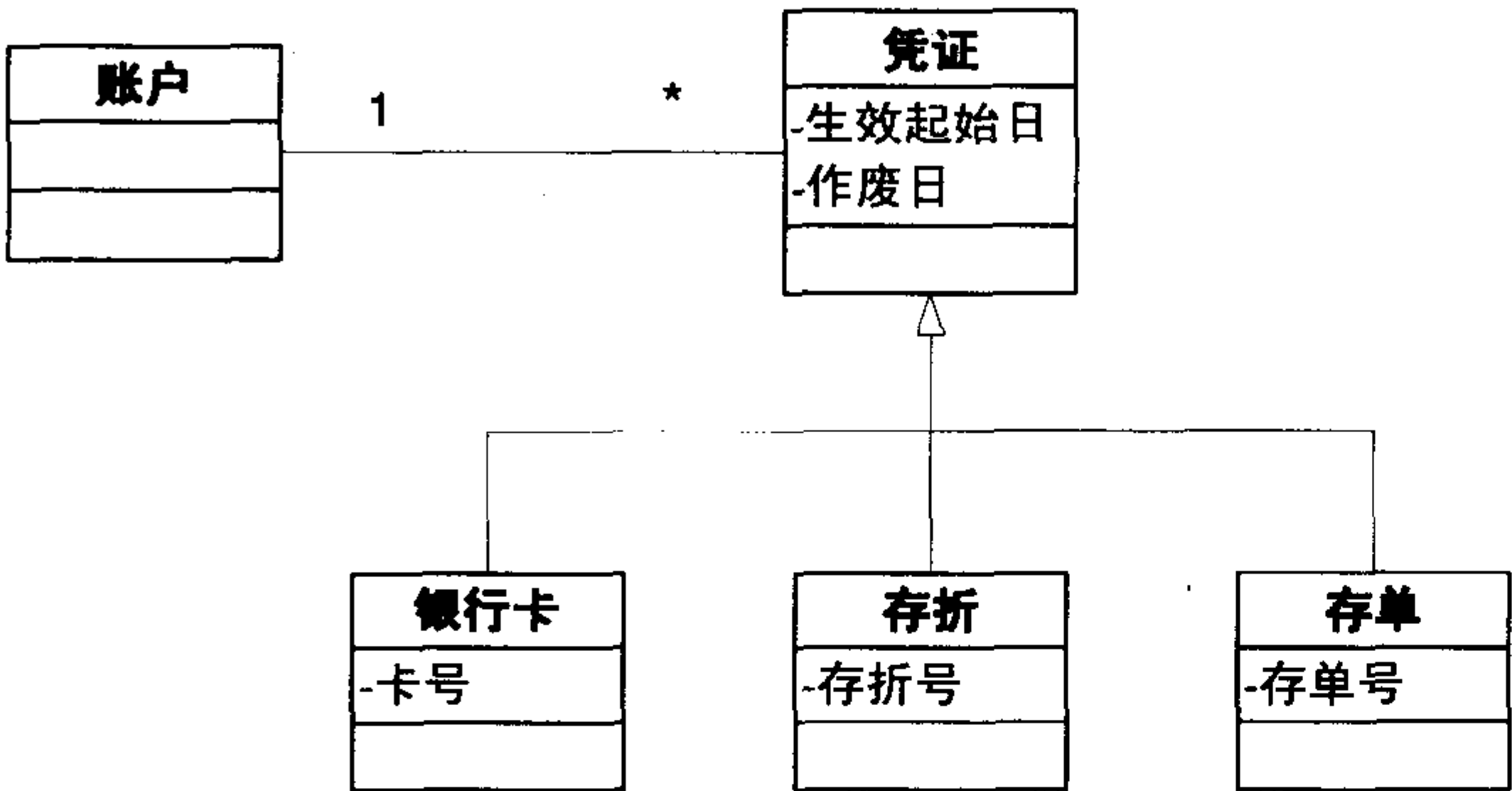


图 12-1 银行领域模型的凭证相关部分

这是一幅 UML 类图，它抽象地表示了银行领域中和凭证相关的部分领域知识：

- 任何一个银行“账户”（这里没有详细分类）可能与多个“凭证”相关；
- 具体而言，凭证可以是银行卡、存折或存单等形式；
- 任何凭证都有明确的生效起始日和终止日；
- 但各种凭证的凭证号却不是统一的，比如存折和信用卡有不同的编号格式；
- .....

模型虽小，却涵盖了银行一些实际的业务情况。由此例可以看出：领域模型是对实际问题领域的抽象表示，它专注于分析问题领域本身，发掘重要的业务领域概念，并建立业务领域概念之间的关系。

Grady Booch 在《面向对象项目的解决方案》中说明了“关于一般领域模型规模的经验性法则”，这一法则无疑为我们提供了更多领域模型的实感：

对于中等复杂度的项目，应该在系统的领域模型中找到大约 50 ~ 100 个类，它们只代表定义问题空间词汇的那些关键抽象。

12.1.2 领域模型相关的 UML 图

一般情况下，领域模型用下面两种 UML 图表示：

- 类图
- 状态图

类图无疑是用得最多的，但有时状态图可以用来对业务领域对象的状态变化进行有效的补充说明。仅举一例。图 12-2 描述了储蓄账户的可能状态及状态转换关系。

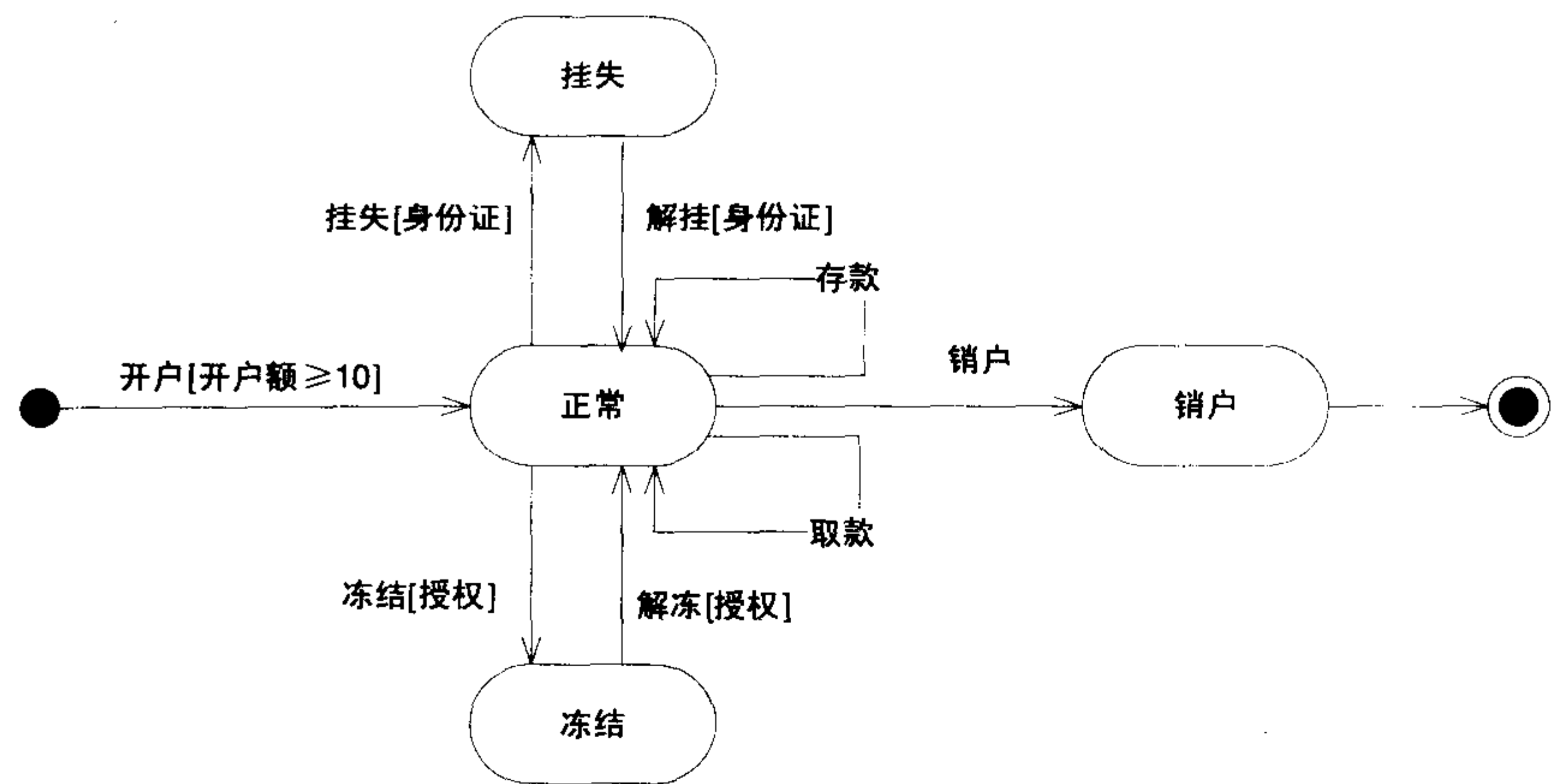


图 12-2 储蓄账户的可能状态及状态转换关系

该状态转换图作为银行领域模型的一部分，表达了如下业务知识：

- 储蓄账户有正常、挂失、冻结和销户等 4 种状态；
- 有效的储蓄账户始于开户交易，开户交易成功后储蓄账户处于正常状态；
- 开户交易的业务规则是：开户金额 $\geq 10$  元人民币；
- 用户可以凭身份证要求对自己的储蓄账户进行挂失和解挂交易；
- 银行可以根据授权（例如司法授权）对储蓄账户进行冻结和解冻；
- 处于正常状态的储蓄账户可以进行存款、取款交易；
- 处于正常状态的储蓄账户经销户交易后变成销户状态。

## 12.2 领域建模在软件过程中所处的位置

成功的项目都是相似的，失败的项目却各有各的失败之处。对于采用面向对象技术的软件项目而言，领域建模恰恰属于“都是相似的”之列——领域建模是公认的促使 OO 项目成功的最佳实践之一。

### 12.2.1 领域建模的必要性：从需求分析的两个典型困难说起

有经验的需求分析员会特别注意避免常见问题的发生，下面我们来讨论其中的两个。

第一个困难：用户的参与不够，造成需求分析成果中假设的成分太多。很典型的情况是，用户不能理解开发方为什么要投入如此多的精力进行需求捕获和需求分析，在他们看来，“需求很

明白”；另一方面，在用户真正使用了软件系统一段时间之前，他们往往并不确切地知道自己需要什么。这是一个悖论，除了运用原型等方法进行需求启发之外，我们有必要和用户进行“更深一级”的沟通。

领域模型是对实际问题领域的抽象，它“穿透”用户想要的功能的表象，专注于分析问题领域本身，发掘重要的业务领域概念，并建立业务领域概念之间的关系。因此，开发方和用户在“领域模型”上达成的共识，往往比在“功能需求”上达成的共识“更深一级”，从而也更加稳固。

可以这么说，“用户参与不够”其实是两个问题：用户参与不够多，或者用户参与不够深入。

需求讨论、原型评审和现场客户等方法可以比较好地解决“用户参与不够多”的问题。通过和用户一起进行领域建模的讨论、领域模型的评审，可以比较好地解决“用户参与不够深入”的问题。用户代表或领域专家深入参与领域建模活动，可以避免需求分析员对业务领域的理解一直停留在假设的层面，直到软件开发出来后才被用户发现的尴尬局面出现。

第二个困难：问题领域太复杂时，需求分析的开展会遇到困难。一个广为人知的案例来自敏捷社区，《敏捷软件开发生态环境》中记载了一个因“分析瘫痪”而惨败的案例：

新加坡贷款项目是一个巨大的失败。在 Jeff 没有加入该项目之前，一家知名的大型系统集成公司（在此当然不便点名）在该项目上花了两年时间，最后宣布做不下去了。……

该项目是一个涉及大范围的商业、公司和消费者的贷款系统，它融合了大量的贷款凭证（从信用卡到大型跨银行的公司贷款）和广泛的贷款功能（从调查到完成到后台监测）。……

而 Jeff 却成功了，他的成功来自许多最佳实践，其一就是领域建模。《敏捷软件开发生态环境》中继续写道：

进入到“新”项目不到两个月时间，Jeff 的团队就开始向客户提供可演示的特性。该团队花了大约一个月时间进行整体对象模型方面的工作……

类似下列情况你也许并不陌生。需求分析在进行过程中，我们可能不断地因“对关键领域问题的理解不足”而卡壳或者争论不休。例如，银行系统中客户、账户、凭证的关系因“一本通”、“一卡通”的出现变得复杂了，需求讨论可能一而再、再而三地影响需求分析的推进。

怎么办呢？可以借助于领域建模。柳传志曾说过，做事要“撒上一层土，夯实了，再撒上一层土，再夯实了”，很有借鉴意义。对于需求分析而言，也存在一个领域知识的“夯实”问题。借而鉴之，我们在需求分析过程中不应忘记：搞清楚一部分领域知识，就将此部分知识建模并将模型在整个项目组公开，再搞清楚一部分领域知识，再建模并将模型在整个项目组公开。

通过上面“两个困难”的讨论可以看出，领域建模对需求分析起着必要的支持作用。



### 12.2.2 领域建模与需求分析的关系

领域模型是探索问题领域的工具，可以帮助我们探索和提炼问题领域知识。那么，它和需求分析是什么关系呢？从本质上来讲，领域建模和需求分析活动是相互伴随、相互支持的。如图 12-3 所示。一方面，领域模型提供的词汇表应当成为所有团队成员所使用语言的核心，在需求活动以及其他活动中起到团队交流基础的作用。另一方面，需求捕获和分析非常关键，因为不知道客户想要什么，就无法提供让客户满意的软件。

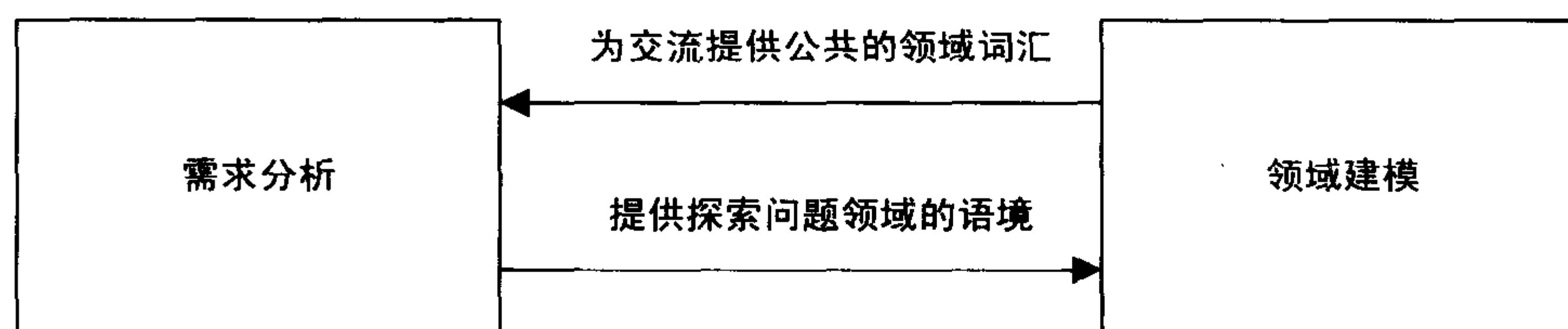


图 12-3 领域建模与需求捕获之间的关系

关于是先有领域模型还是先有需求定义的问题，在业界也可算一个颇有争议的话题了。其实，务实来讲，它们是同时产生、交叠演进的。图 12-4 展示了常见的项目过程是如何规划的。

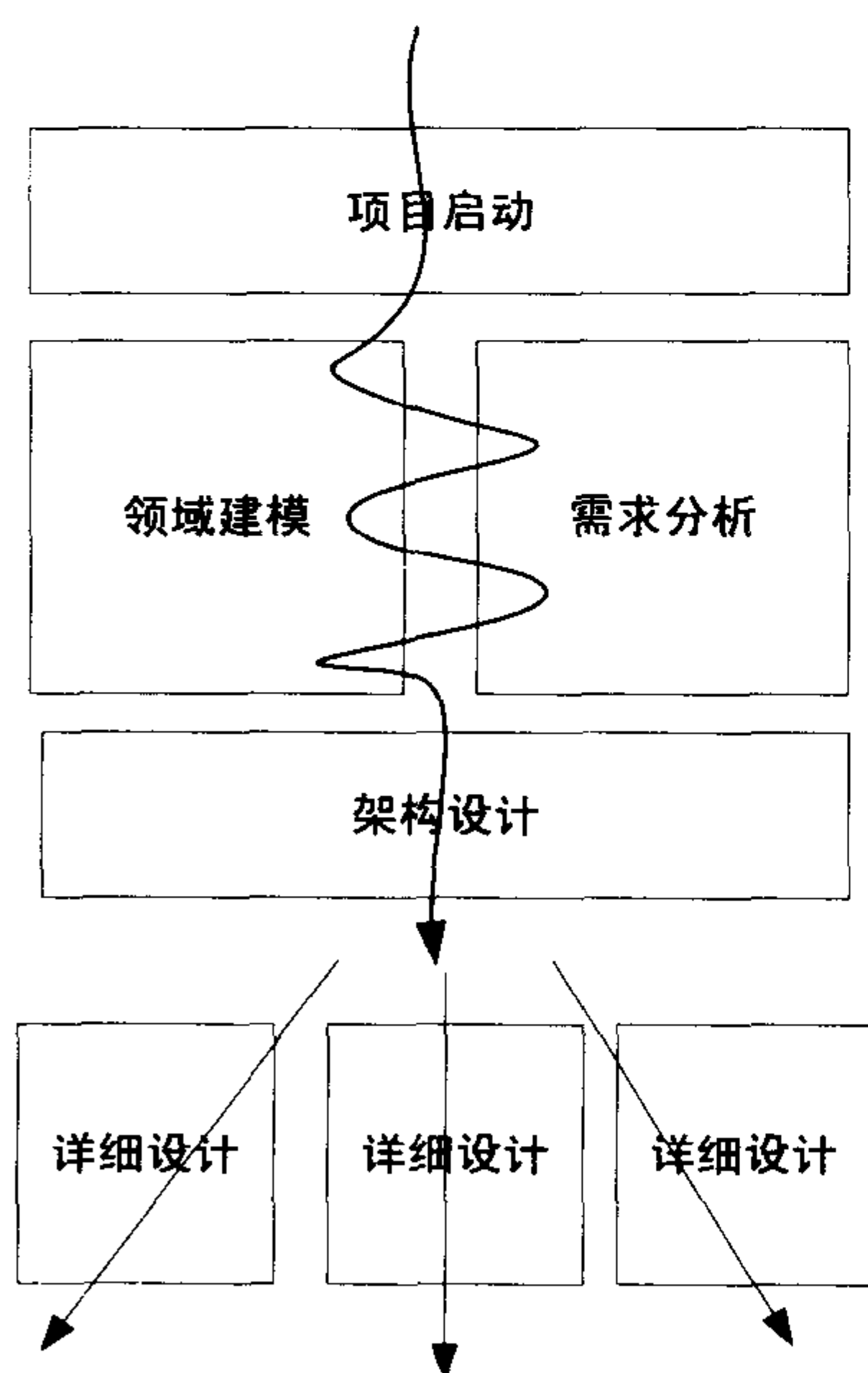


图 12-4 常见的项目过程是如何规划的

12.2.3 领域建模所处的位置

图 12-5 总结了领域模型对整个软件开发活动的重要作用：

- 领域模型为需求定义提供了领域知识和领域词汇。较之《词汇表》而言，领域模型更能体现各领域概念之间的关系，有较好的大局观；
- 软件界面的设计往往和领域模型关系密切。一方面，领域信息是所要展现内容中最重要的部分；另一方面，界面结构必须和业务内容的结构相一致，否则会使软件晦涩难用；
- 领域模型是否合理将严重影响软件系统的可扩展性。这是因为，丰富多彩的软件功能背后“藏”着的领域模型决定了软件功能可能的范围。Martin Fowler 在《分析模式》一书中指出：模型的选择会影响最终产生的系统的灵活性和可重用性。此言不虚；
- 由于分层架构的思想被广泛接受，领域模型经过精化之后会成为业务层的核心；
- 领域模型是设计持久化数据模型的良好基础，在实践中直接将领域模型映射成物理数据模型的例子也屡见不鲜。

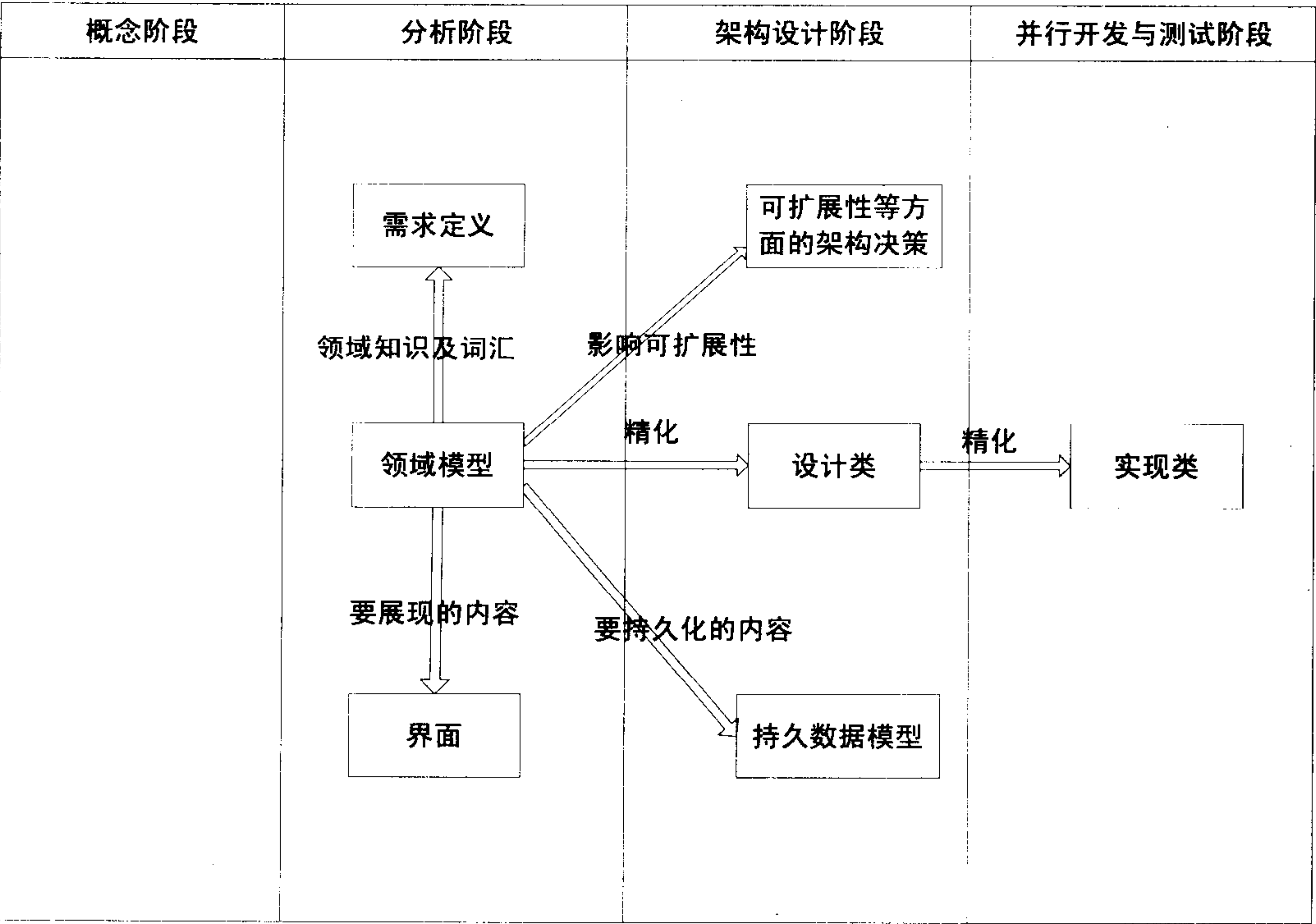


图 12-5 领域模型在软件开发中的作用

## 12.3 领域模型对软件架构的重要作用

软件的架构设计对整个软件开发任务而言是举足轻重的，而领域模型对架构设计的成功与否又起着非常关键的作用。总体而言，领域模型对软件架构乃至整个软件系统开发工作的作用可以归纳为如下 3 点：

- 探索复杂问题、固化领域知识
- 决定功能范围、影响可扩展性
- 提供交流基础、促进有效沟通

下面结合案例分别进行讲解。值得说明的是，为了使没有接触过领域建模的读者尽快熟悉领域建模，我们决定引入多个实际案例——在说明领域模型的每个作用时引入一个新的案例。

### 12.3.1 配置管理工具案例：探索复杂问题、固化领域知识

《人月神话》一书指出了软件开发的“根本问题”，所谓根本问题，就是无论如何也回避不掉的问题，首当其冲的就是“软件的复杂性”。而运用面向对象的领域模型技术，可以帮助我们驯服复杂的问题。对此，Grady Booch 和 Martin Fowler 都不约而同地表示过，他们采用面向对象方法最大的原因是它有助于解决更为复杂的问题。领域模型本身作为辅助思维的工具，帮助我们注意力始终保持在最为重要的业务概念及其关系上，使我们能够不断深入地、系统地整理对问题的认识。

下面举例。

笔者曾经负责软件配置管理工具的分析和设计工作。由于软件配置管理领域有不少非常专业的术语，并且配置管理和整个软件开发过程有着密切的支持和被支持的关系，因此使得配置管理领域比较复杂。在这种情况下，项目伊始，通过领域建模来探索并理清复杂的问题，是非常有实际意义的实践方法。

领域建模往往要经历一个从模糊到清晰，从零散到系统的过程。例如，图 12-6 展示了对软件配置管理领域模型的早期的、零散的、不完整的认识。

从此图中可以看出，负责领域建模的架构师在配置管理领域专家的帮助下，已经抓住了一些有价值的知识：

- 软件配置管理中，被管理的对象不是抽象的概念，而是实实在在的以“工件”形式存在的工作成果；
- 随着软件开发和管理工作的开展，工件可能发生“变更”——或被增强、或被修改，不一而足；
- “基线”和“配置项”都是软件配置管理中非常关键的概念……



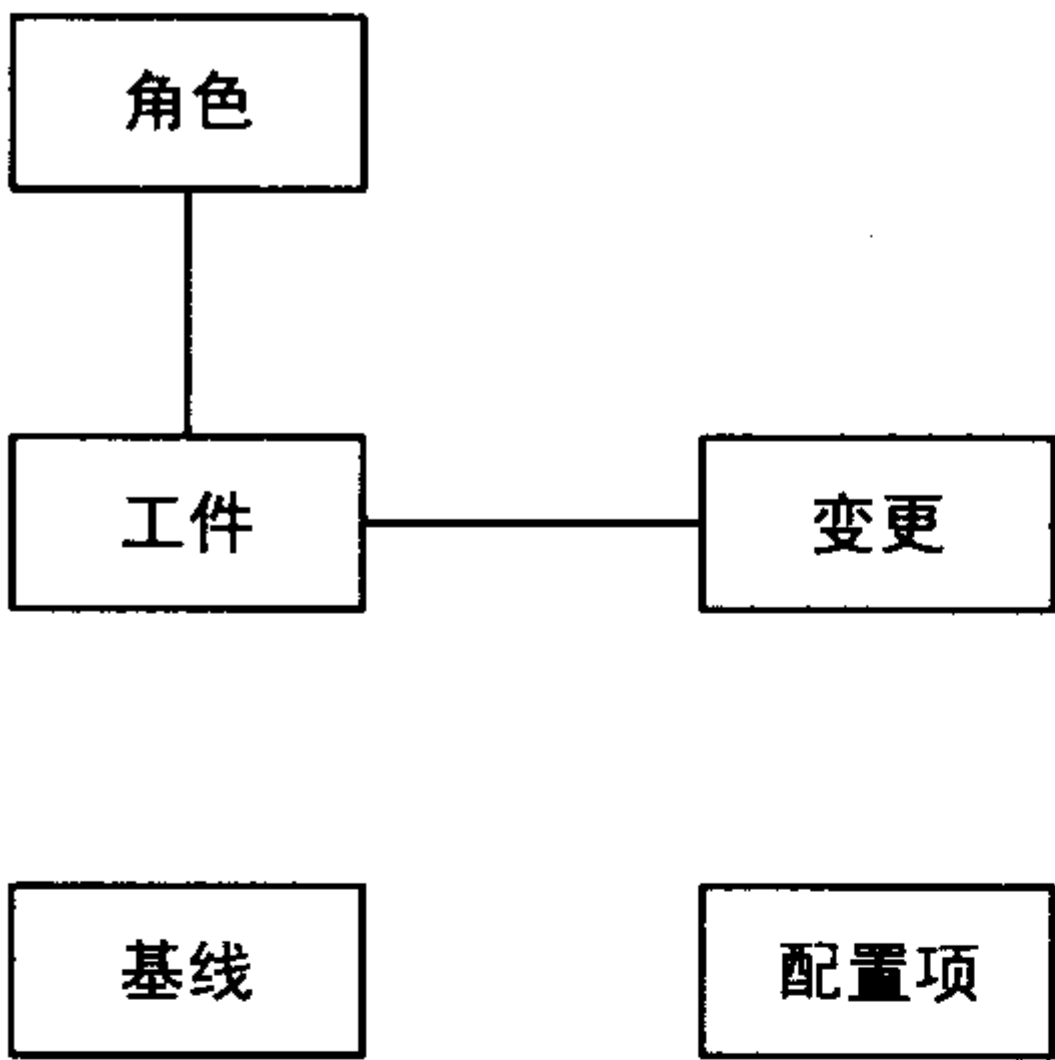


图 12-6 开始时的领域模型可能零散且不完整

软件架构师和领域专家通力合作，在稚嫩却有启发性的最初领域模型的基础上不断讨论、不断探索、不断挖掘，使领域模型不断精化，最终的领域模型变得越来越严谨、越来越系统。图 12-7 展示了最终软件配置管理领域模型的一部分。

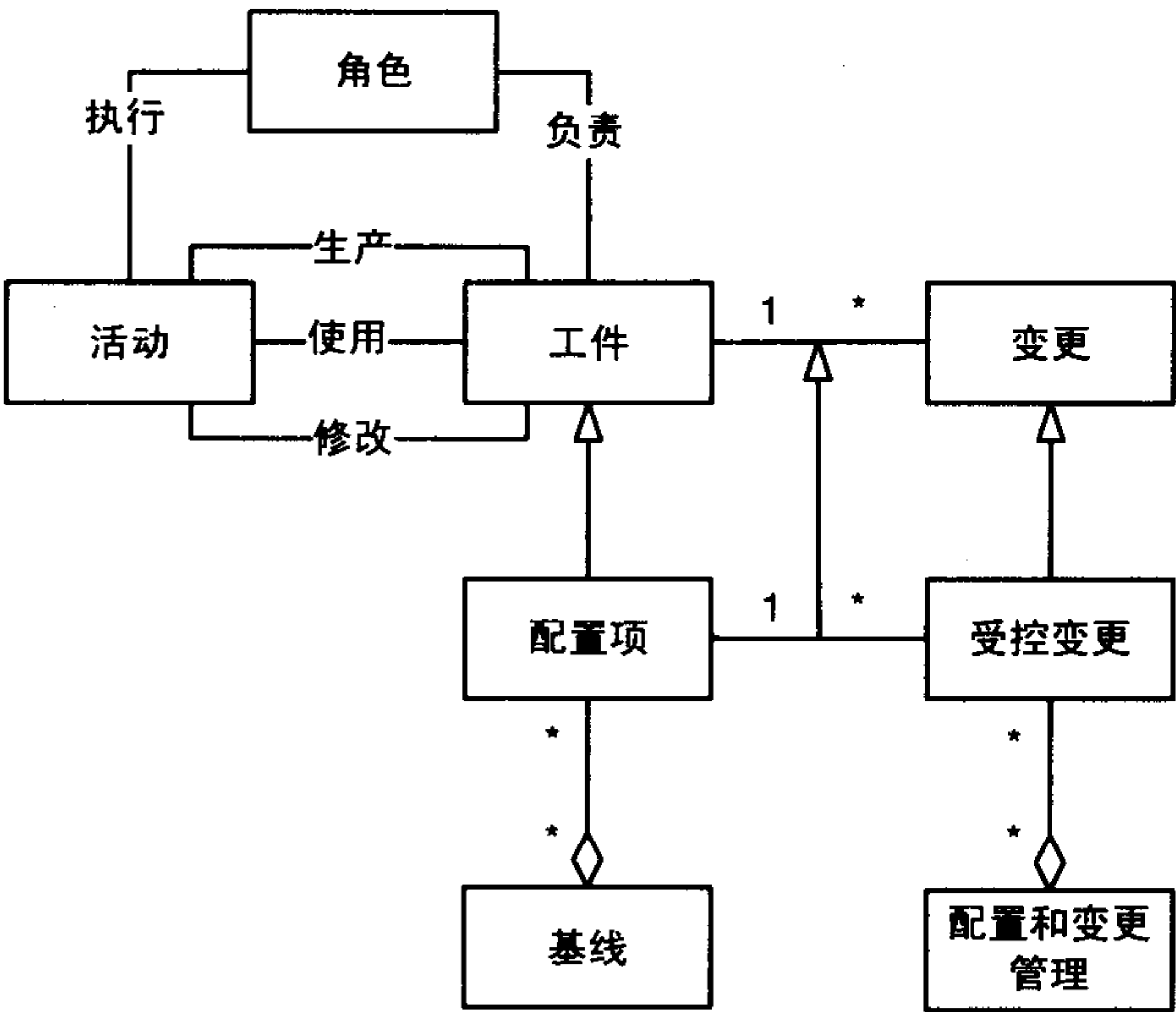


图 12-7 软件配置管理领域模型（部分）

任何软件工程过程，都少不了角色（role）、活动（activity）和工件（artifact）等概念（或者类似概念）。这些概念本身很好理解。角色是对个人或者作为开发团队的一组人的职责的规定；具体人和角色的关系，好比人和帽子的关系。活动就是角色执行的工作单元。工件就是工作的成品或半成品。

倒是这些概念的关系显得更加重要。角色的职责，具体体现在他执行的活动和负责的工件上。工件是由活动生产出来的——工件是活动的输出；比如制定《编码规范》。然而，活动本身也可能以工件为输入——活动可能要求使用工件；比如编码活动要参考《编码规范》。还有一种

关系，工件既是活动的输入又是它的输出——活动修改工件；比如修改《编码规范》。

建立并管理基线（baseline），为软件开发管理提供了有力的支持。基线的要点有二：一是要通过评审，二是要受配置和变更管理控制。IEEE 对于基线的完整定义是：已经通过正式复审和批准的某规约或产品，它因此可以作为进一步开发的基础，并且只能通过正式的变更控制过程进行改变。

配置和变更管理完成建立并管理基线的任务。置于配置和变更管理之下的工件，被称为配置项（configuration item）——因此模型中把配置项建模为工件的子类。而基线就是由多个配置项组成的瞬时快照——因为受配置和变更管理控制是基线的必要条件。任何工件都有可能发生变更；正如并不是所有工件都是配置项一样，变更也不一定都受控，比如，用于讨论的临时设计就不必受控。只有配置项的变更，才是需要受配置和变更管理控制的。到底哪些工件应当受控，是根据实践情况决定的，应当在规范性和灵活性之间权衡考虑。

例子讲完了。

通过对案例的分析，我们看到领域建模的过程就是对复杂问题进行探索的过程，最终得到的领域模型将复杂的领域知识记录下来、“固化”下来。

领域建模活动的结果是领域模型，但是由于问题领域的复杂性，领域模型不是一蹴而就的。在领域建模过程中，前期的成果往往很不完善，我们会不断基于现有的成果进行讨论、分析和完善。因此，领域建模的过程就是探索复杂问题的过程，而领域模型的早期版本本身也成了促进进一步思维的工具。

12.3.2 人事管理系统案例：决定功能范围、影响可扩展性

为了说明领域模型如何决定功能范围并影响软件系统的可扩展性，我们引入一个新的案例（或许这个案例正是你熟悉的领域）。

这是一个人事管理系统的例子。

客户对最初交付的 HR 管理系统他们比较满意。在这个版本中，架构师根据“统计公司雇员”等需求（没有考虑可扩展性）设计了领域模型，图 12-8 展示了这个领域模型的一角。

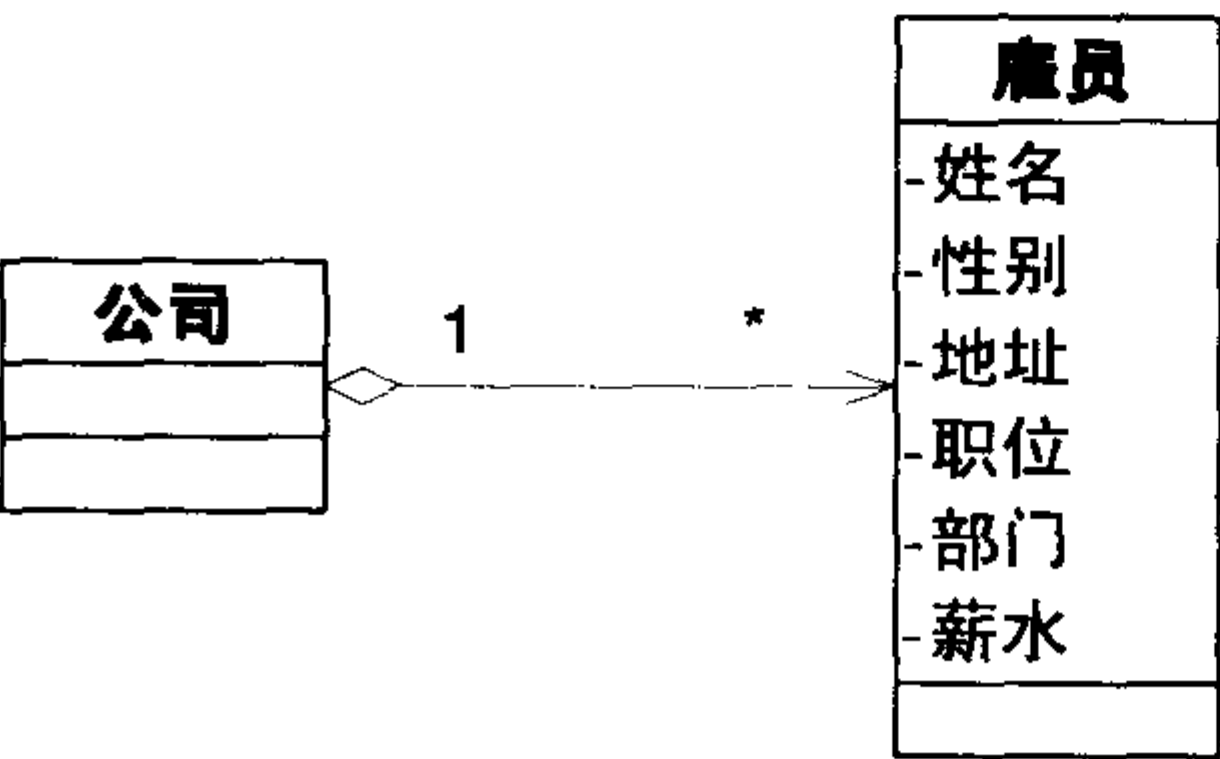


图 12-8 最初的人事管理系统领域模型之一角

时间流转，公司开始出现职位升迁、员工离职，甚至离职的员工又被“挖”回来等情况。这时，HR 管理系统出现了问题，比如 HR 经理希望列出如图 12-9 所示的员工履历纵览，但系统却只能显示某员工的最新职位。


员工信息			
姓名	性别		
地址			
员工履历			
时间	部门	职位	薪水
— —	— —	— —	— —
— —	— —	— —	— —
— —	— —	— —	— —
— —	— —	— —	— —

图 12-9 最初的模型不支持此功能

毋庸置疑，必须对系统进行升级才能支持“能够反映职位变动和离职员工重新加入公司”这一特性（Feature）。但这时我们发现，由于领域模型的限制，希望仅扩充应用层（相当于领域层、数据层）就增加对“查看员工履历”功能的支持是不可能的。这是因为领域层并没有提供“查看员工履历”所需的服务，而且领域层也没有能力提供出这样的服务。

没办法，领域模型必须被重新设计以增强能力，如图 12-10 所示。说明一下，由于建模者的习惯差异，有的人常用“关联类”而有的人不用。图 12-10 中，分别用两种方式表达了相同的设计思想，只不过左边的模型采用了关联类语法而已。

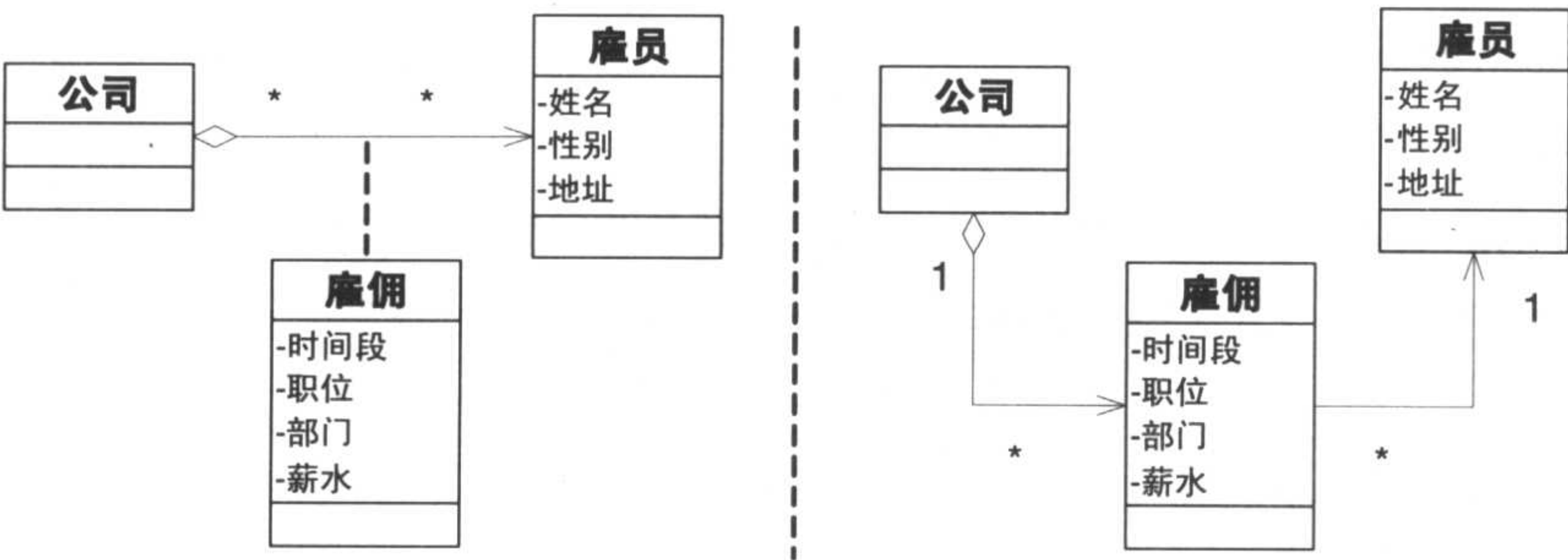


图 12-10 升级后的模型（前者采用关联类）

不难看出，公司和雇员原来是一对多关系，现在变成了多对多关系；更深层次的，职位、所在部门、薪水这些属性从“雇员类”移到了“雇佣类”。分析如下：

- 雇员的姓名、性别很少改变（或者说，本 HR 系统不打算支持那些极端情况），而地



址只需记录最新的即可，因此姓名、性别、地址作为雇员的属性；

- 和姓名不同，雇员的职位、部门、薪水并不是一成不变的，而是随着职业生涯的发展不断变化的。从面向对象的角度考虑，职位、部门、薪水不应作为雇员的属性；
- 其实，公司雇佣了某人作为雇员，那么公司和雇员之间就建立了雇佣和被雇佣的关系，而职位、部门、薪水都是对这种雇佣关系的描述。从面向对象的角度考虑，可以采用关联类来描述“雇佣”这种关系。除了职位、部门、薪水等属性之外，雇佣类还应该担负记载雇佣时间段的职责。

这样一来，为了支持“能够反映职位变动和离职员工重新加入公司”这一特性（Feature），不仅具体实现要增强、数据库要改变，而且子系统的接口也会随之发生改变，使这一变化的影响波及到其他子系统。

下面，让我们仔细体会一下上面例子所说明的问题。

变化无处不在。一个软件系统，用户交互的方式变换花样了，持久数据的存储从文件方式变成关系数据库（甚至对象数据库）了，或许还和形形色色的其他系统“无缝集成”了……

但并非变化无常（“常”者，规律也）。比如，问题领域层的对象似乎在某种程度上还是“老面孔”。关于这一点，一个老于世故的例子是航空订票系统——从架在“大型主机+字符终端”之上的基于命令行的订票系统，到现在的 B/S 结构的分布式订票系统，“航班”、“乘客”及“优惠策略”这些领域概念似乎从未变过。

领域模型决定了软件系统功能的范围。任何模型都是对现实世界的某种程度的抽象，领域模型也不例外。抽象意味着有目的地忽略；而忽略哪些对象和关系，保留哪些对象和关系，都和具体目的有关。最初的人事管理领域模型（见图 12-8）可以支持很多有用的功能，但这个模型不能反映雇员的职位和部门等的变化历史。后来的人事管理领域模型（见图 12-10）更为复杂，提供了记录历史的功能。

领域模型还影响着软件系统的可扩展性。可扩展性是软件系统的一种非功能性的质量属性，是在系统保持现有功能特性的基础上，扩展实现有一定相关性的其他功能特性的容易程度。扩展越容易则可扩展性越高。功能是软件系统能够完成的具体任务；而模型揭示了丰富多彩的功能需求背后的结构，使我们设法“驯服”复杂性时有了切中要害的“着力点”。如果说，定义系统的功能相当于“拍照片”的话，那么领域建模就相当于“做透视”：功能需求记录了系统外在的、用户可感知的“应该列表”，但它们是易变的，当商业环境急剧变化的时候需求尤其易变；而领域模型揭示并模拟问题领域的内在结构，相当于对问题领域进行了一层抽象，良好的领域模型不仅支持现有功能需求的满足，还在一定程度上支持未来可能出现的新需求，使系统需要扩展的时候仅需增加必要的“应用功能代码”，体现良好的可扩展性。

12.3.3 在线拍卖系统案例：提供交流基础、促进有效沟通

有人说，团队开发最重要的问题有三个：交流，交流，还是交流。Eric Evans 在《领域驱动设计》中揭示了我们普遍认识到的问题：

领域专家对于软件开发的技术行话的理解通常都非常有限，但是他们会使用他们自己领域的行话——很可能有很多种“风味”。而另一方面，开发人员能够理解并使用描述性的功能术语来讨论系统，而不使用专家们的语言。或者开发人员可能建立一个能够支持他们设计的抽象结构，而领域专家并不理解。从事于同一问题但不同部分的开发人员也会“搞出”他们自己的设计概念和描述领域的方式。

由于这种语言上的差异，领域专家对他们所需要的只能含糊地进行描述。开发人员费力地去理解领域中对他们来说陌生的东西，也只能含糊地理解领域专家的思想。……

领域模型是团队交流的基础，是所有团队成员所使用语言的核心（如图 12-11 所示）。领域模型规定了重要的领域词汇表，并且这些词汇的定义是严格的、大家共同认可的，所以可以成为团队交流的基础。例如，当开发人员刨根问底地向用户讨教需求、和系统分析员争论需求文档、和测试人员争论那是不是 Bug 的问题之时，就应该用领域模型规定的“领域词汇”，进行不易产生歧义的有效沟通。任何团队开发活动，都应注意交流的问题。团队的不同成员之间共识越多，他们之间的合作就越顺畅。

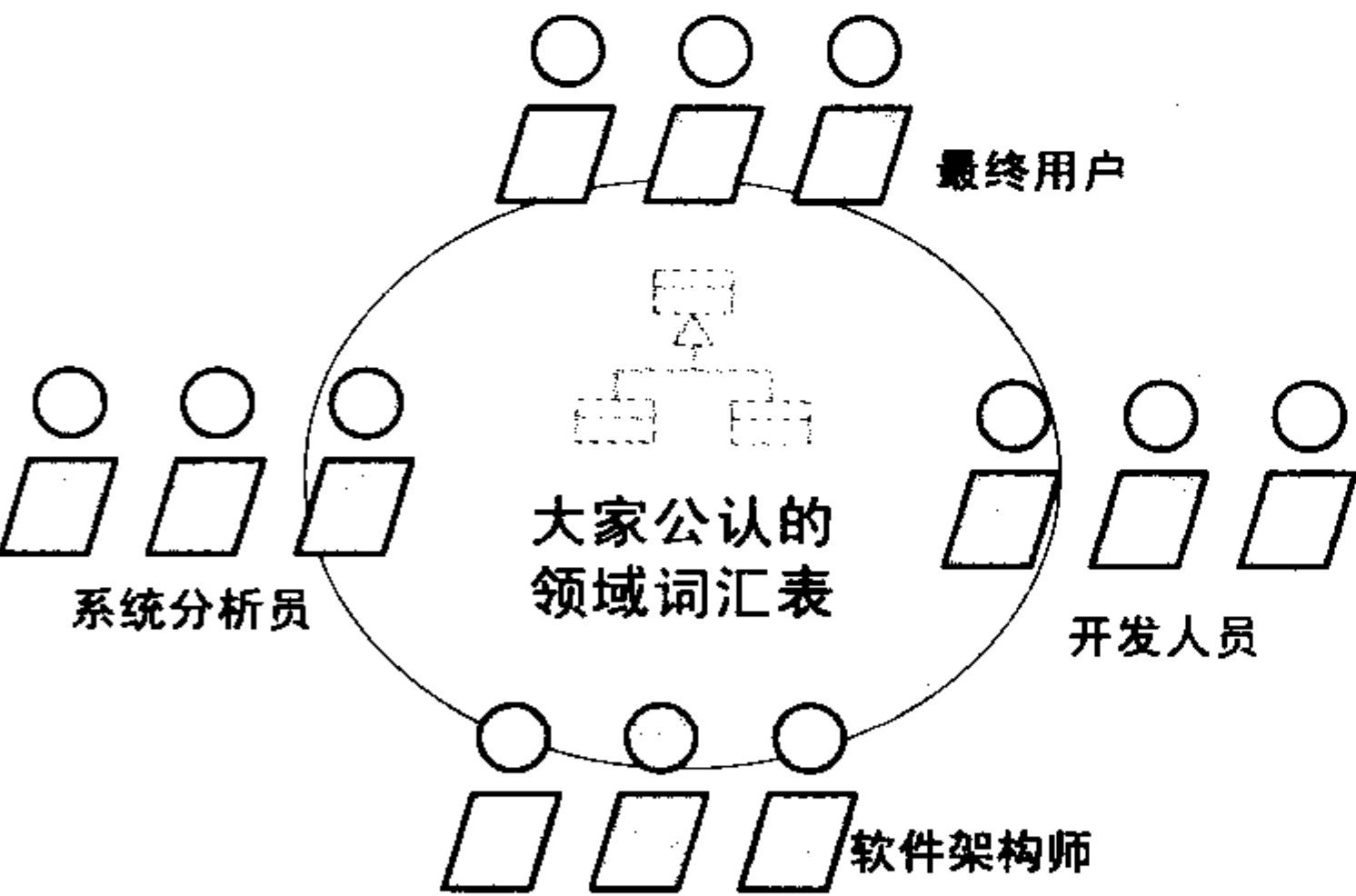


图 12-11 领域模型是团队交流的基础

下面，以类似淘宝网的 C2C 网络平台作为案例，着重讨论“在线拍卖”相关的领域模型，说明其对有效沟通的作用。

拍卖领域的规则比较复杂，图 12-12 和图 12-13 所示为其领域模型的一部分。图 12-12 清楚地表达了和成交相关的领域知识，界定了相关术语，有利于无歧义地进行交流。例如成交是拍卖商品、买家和卖家之间关系的描述，并且其中的买家是“喊出”成交价的那个买家。

然而网上交易涉及到的交付流程比较复杂，图 12-13 对此进行了简化描述。将这些容易理解不清的关键领域知识通过建模“固定”下来，作为交流的基础，就大大降低了交流不畅发生的几率。

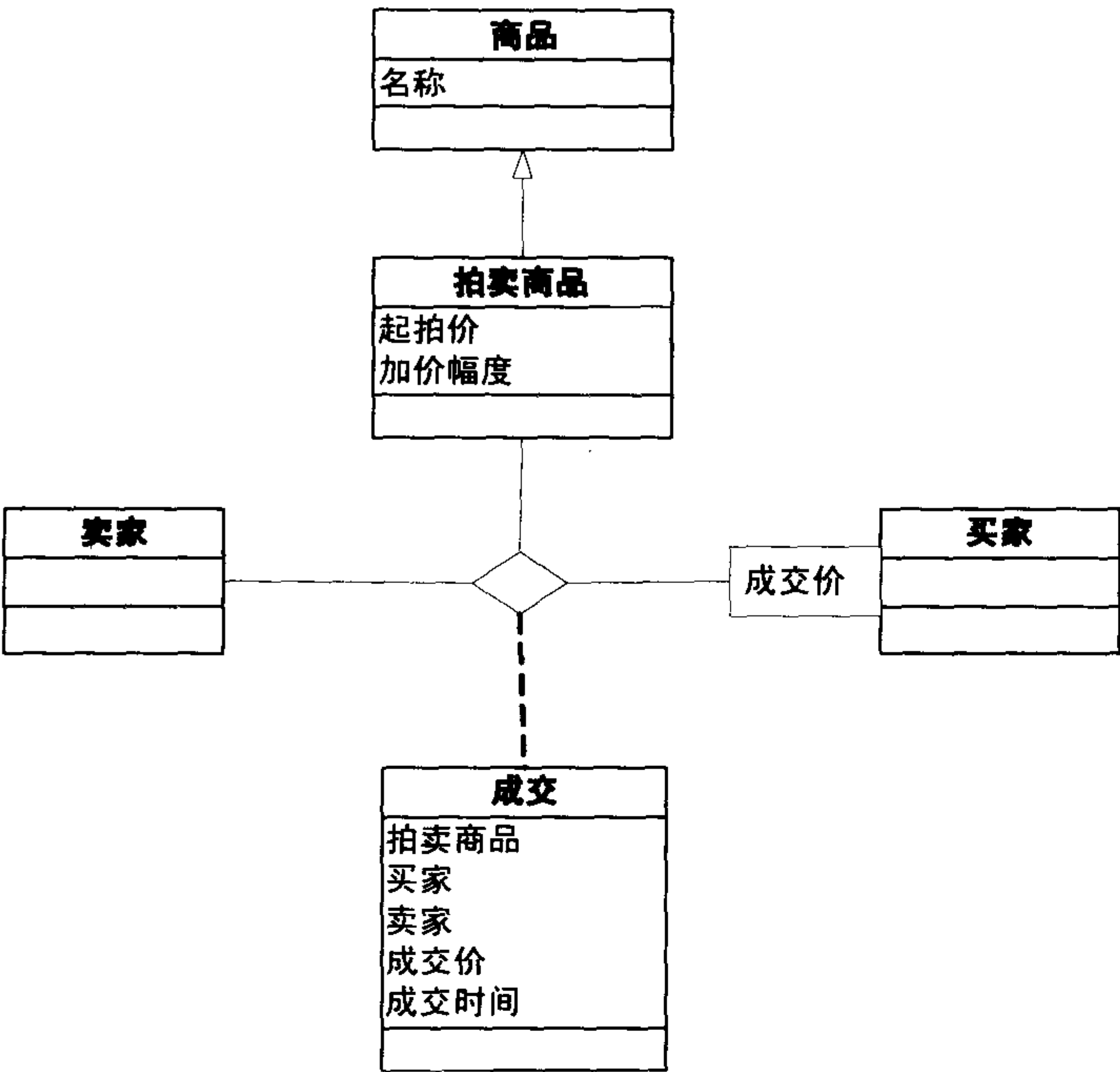


图 12-12 领域模型类图部分

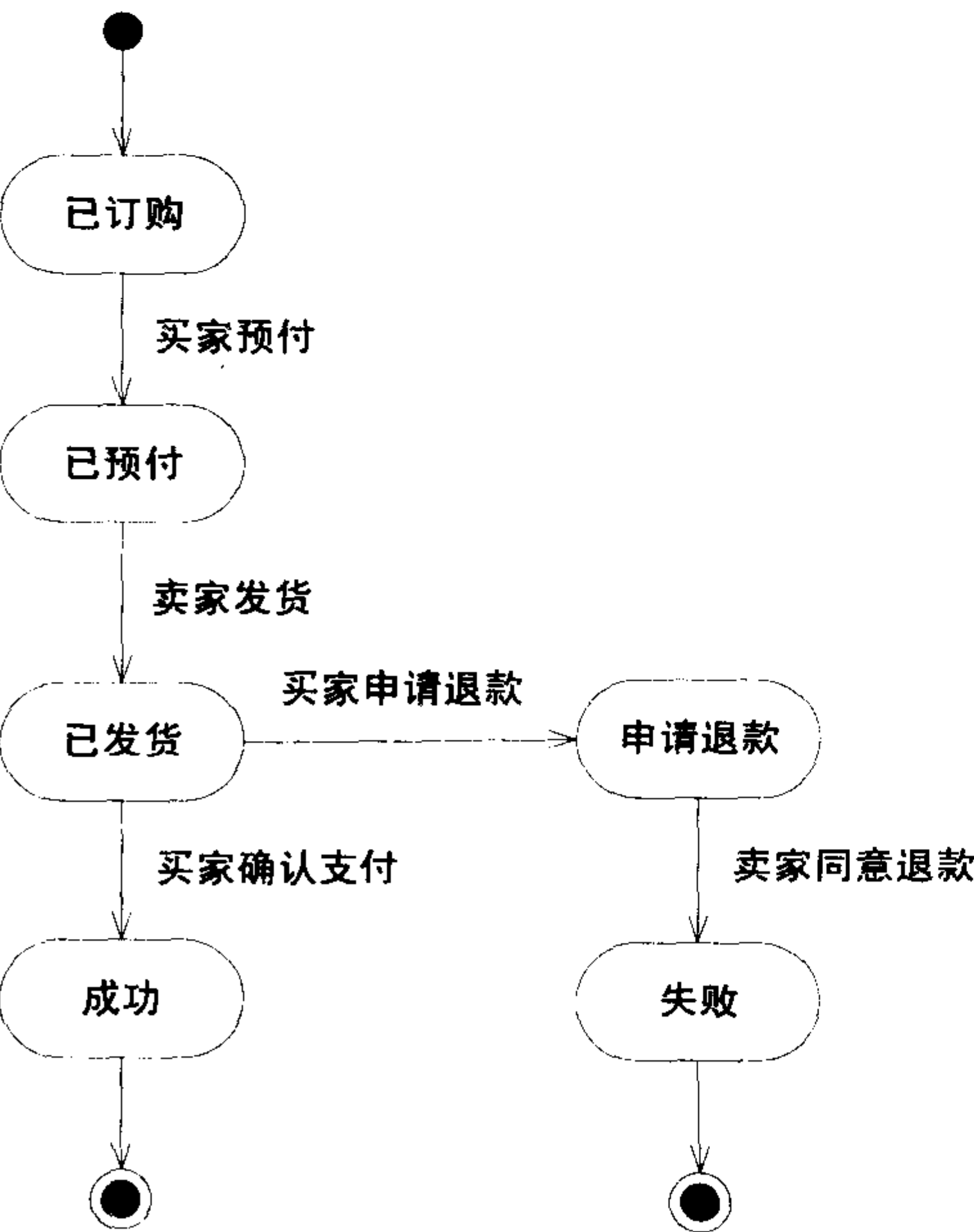


图 12-13 领域模型的状态图部分



## 12.4 领域模型 vs. 文字说明

领域模型和文字说明各有优势，应结合使用。

一方面，建模较之使用自然语言的词汇表，具有“大局观更好”的优点。例如，表 12-1 所示为在线拍卖系统的词汇表，显然不利于揭示众多概念之间的关系，随着领域概念的增多此缺点会更加明显。

表 12-1 在线拍卖系统的词汇表

示例：在线拍卖系统词汇表。
拍卖：一种销售方式，将拍卖项卖给竞拍价格最高者。
拍卖信息：关于拍卖活动的信息，包括拍卖活动的起始时间、延续时间、拍卖项目录等。
拍卖项：进行拍卖的物品的统称。
拍卖项信息：关于拍卖项的信息，包括产品信息（名称、描述、图样、类属）、起拍价、竞拍最小增幅等信息。
起拍价：一开始的竞拍最低价。
成交价：拍卖结束时刻的最高竞拍价。
买家：参与竞买者。
卖家：提供货物，并以拍卖方式出售货物者。
成交：以符合拍卖活动规则的方式，拍卖项的所有权从卖家过让为竞拍价格最高的买家，买家按成交价支付。

另一方面，文字方式自有它的适用场合，而这些场合下建模方式反倒是笨拙的。如表 12-2 所示，为银行卡的卡号规则说明。

表 12-2 银行卡卡号的规则说明

<input type="checkbox"/> 位数：16 位
<input type="checkbox"/> 格式：xxxxx yyyy zzzzzz n
<input type="checkbox"/> 说明：
■ xxxxx 为银行卡组织为各总行卡机构分配的标识号
■ yyyy 为各银行发卡子机构分配的联行号
■ zzzzzz 为发卡序号
■ n 为校验位

笔者认为：UML 模型和词汇表等文字方式应当取长补短、相辅相成。而且有些开发组织（比如一些大型银行）长期使用《词汇表》等方式澄清领域问题，并取得了不错的效果。

# 12.5 PM Tool 实战：建立项目管理的领域模型

## 12.5.1 领域建模实录（1）

领域建模小组正在卓有成效地开展着工作……

领域专家：项目被分解成许多任务，分配下去。

架构师：你是说这个意思吗？（指着图 12-14）

领域专家：正是这样，分工的单位是任务而不是项目。

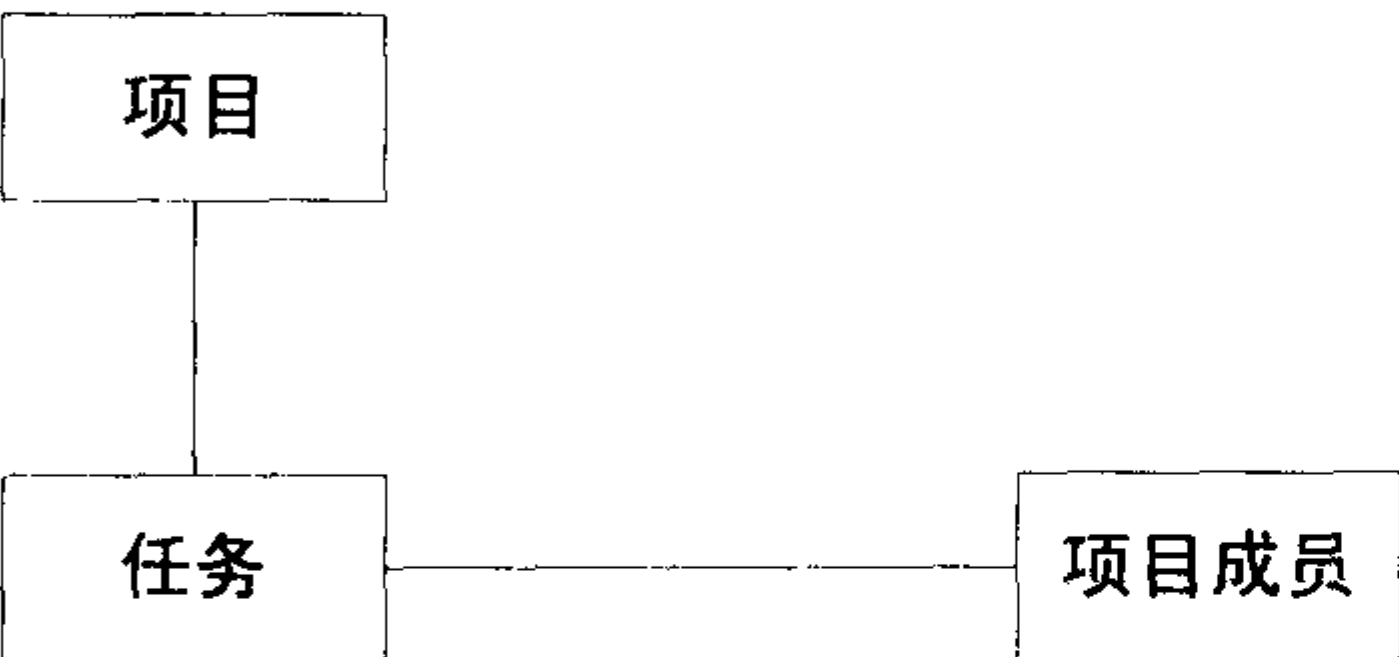


图 12-14 项目分解为任务和项目成员

架构师：也就是说，需要把任务分配给项目成员。（边在图上改着）

领域专家：是的，而且项目规定，每个任务只能分配给唯一的人，而一个人可以负责多个任务。

架构师：好的，这很重要，我要把它们记录下来。（如图 12-15）

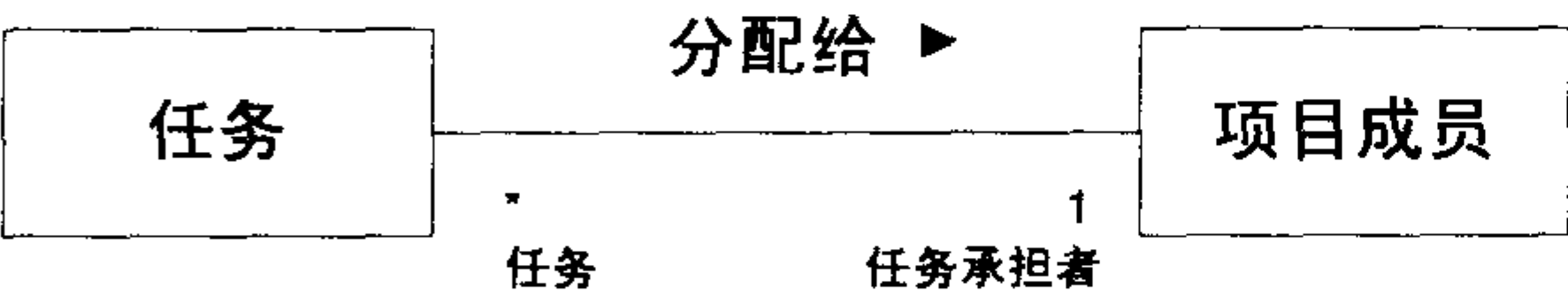


图 12-15 更精确的描述

架构师：恕我啰嗦，一个项目分解为多个任务，但一个任务只能对应于某一个项目。是这样吗？（指着图 12-16）

领域专家：停，似乎有些问题……（盯着图）

领域专家：你听说过“多项目管理”吗？我们就是多项目管理的实践者（骄傲地说）。多个项目可能共享某个子任务。

架构师：多个项目可能共享某个子任务？能举个具体例子吗？

领域专家：比如企业级应用都会用到的一些和领域无关的模块，如身份验证、消息机制等……你是在考我吗？

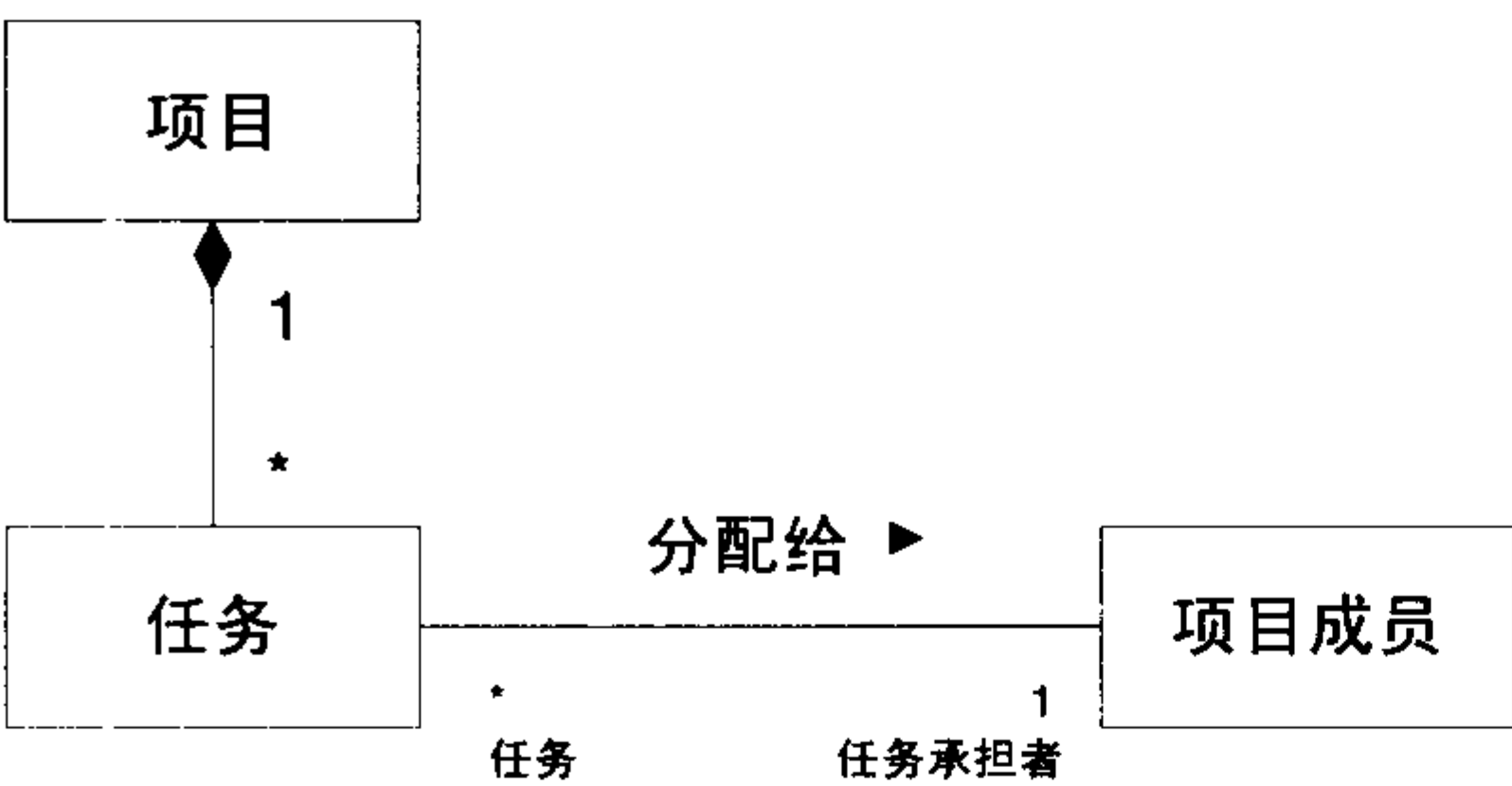


图 12-16 架构师开始认为项目和任务是一对多的关系

架构师：哪里哪里。你看，这就是“不要想当然”的好处，所有我必须能够支持诸如“多项目共享任务”这样的特性。（在图 12-17 上改着）

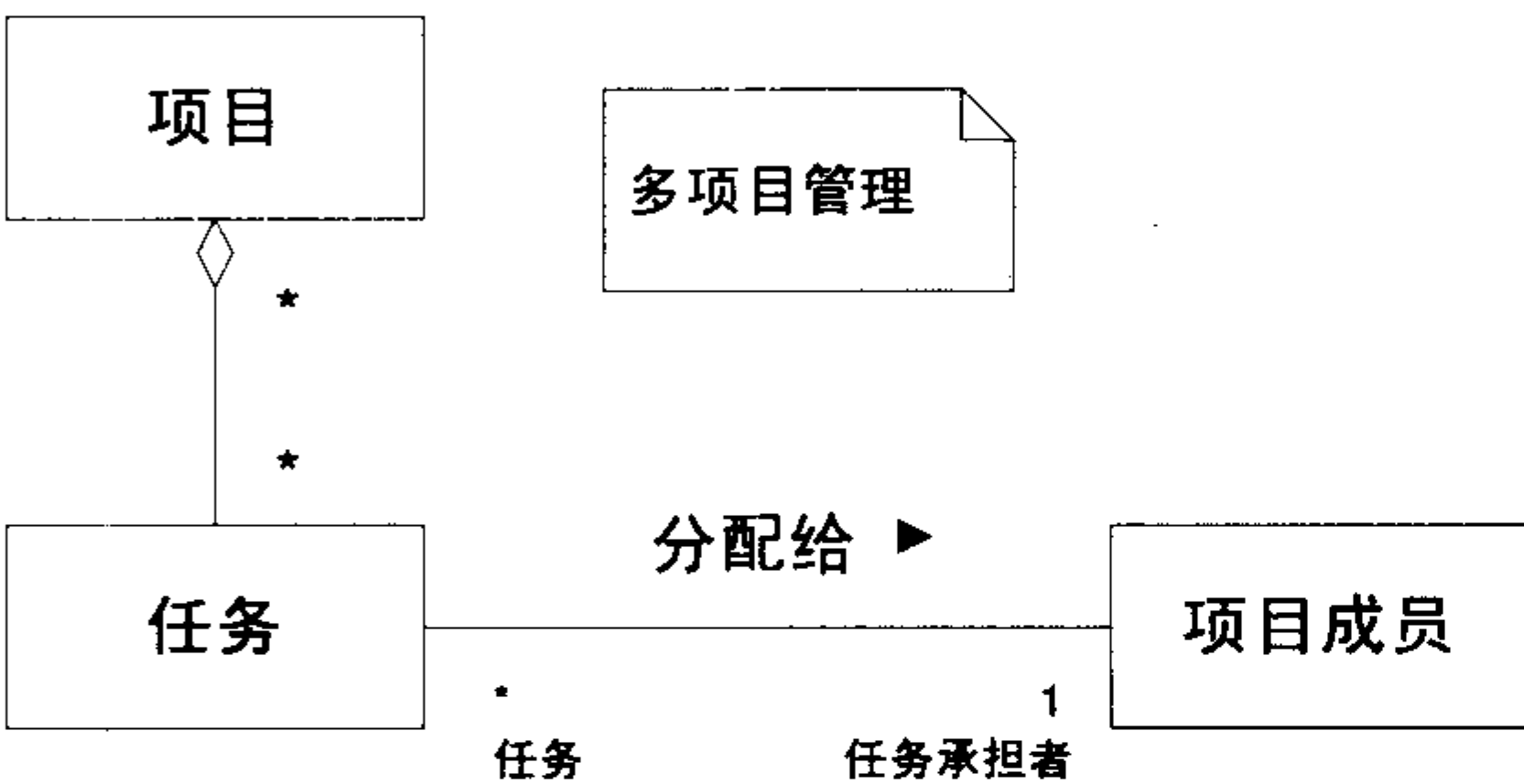


图 12-17 但实际上多个项目可以共享某任务

领域专家：将项目分解为多个任务之后，需要为任务排定日程，比如，某任务结束之后另一任务才能开始……

架构师：哦，很好，任务之间可能有前后关系……（如图 12-18）

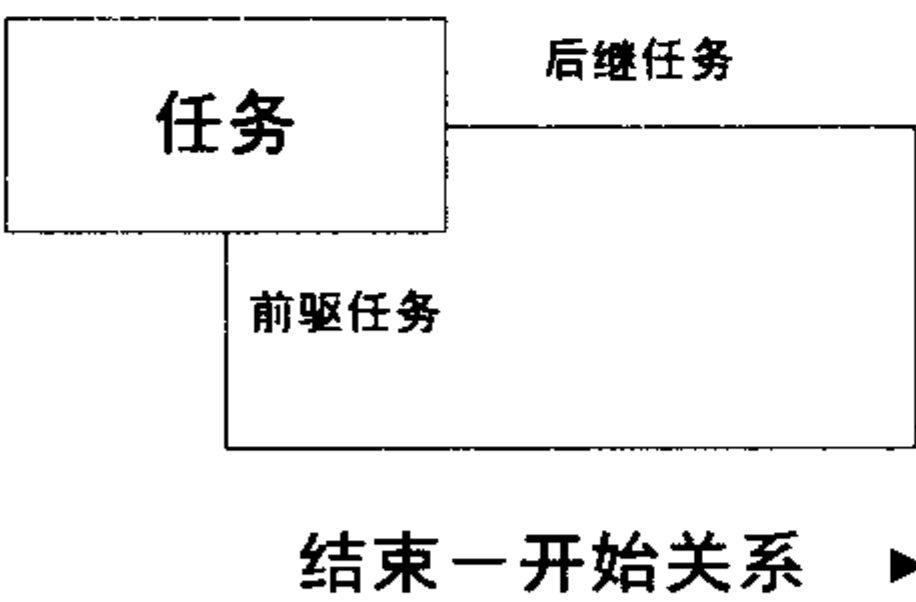


图 12-18 任务之间的前后关系

领域专家：我们的项目涉及的资源有多种，比如人、设备和材料等。没有必要的资源，项目无法开展。

架构师：是的，你桌上放的是仿真器吧，我也曾有过嵌入式系统的开发经验，仿真器是很重要的设备。

领域专家：的确如此。（看着图 12-19 说）



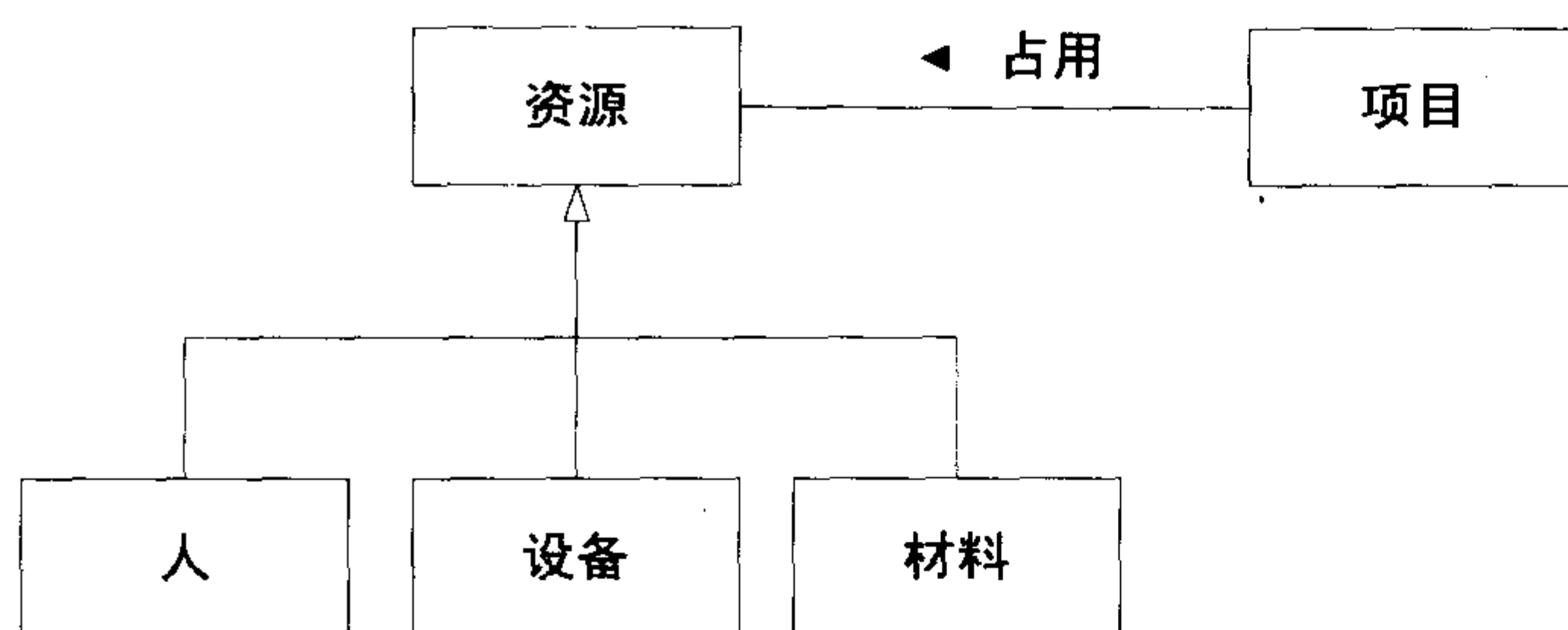


图 12-19 项目占用资源的分类

领域专家：从逻辑上看，资源是项目占用的，但对资源的使用要具体分配到任务，而且应明确具体的使用期限……

架构师：是任务消耗资源，很好。项目、任务、资源的关系越来越清晰了，不是吗？（指着图 12-20）

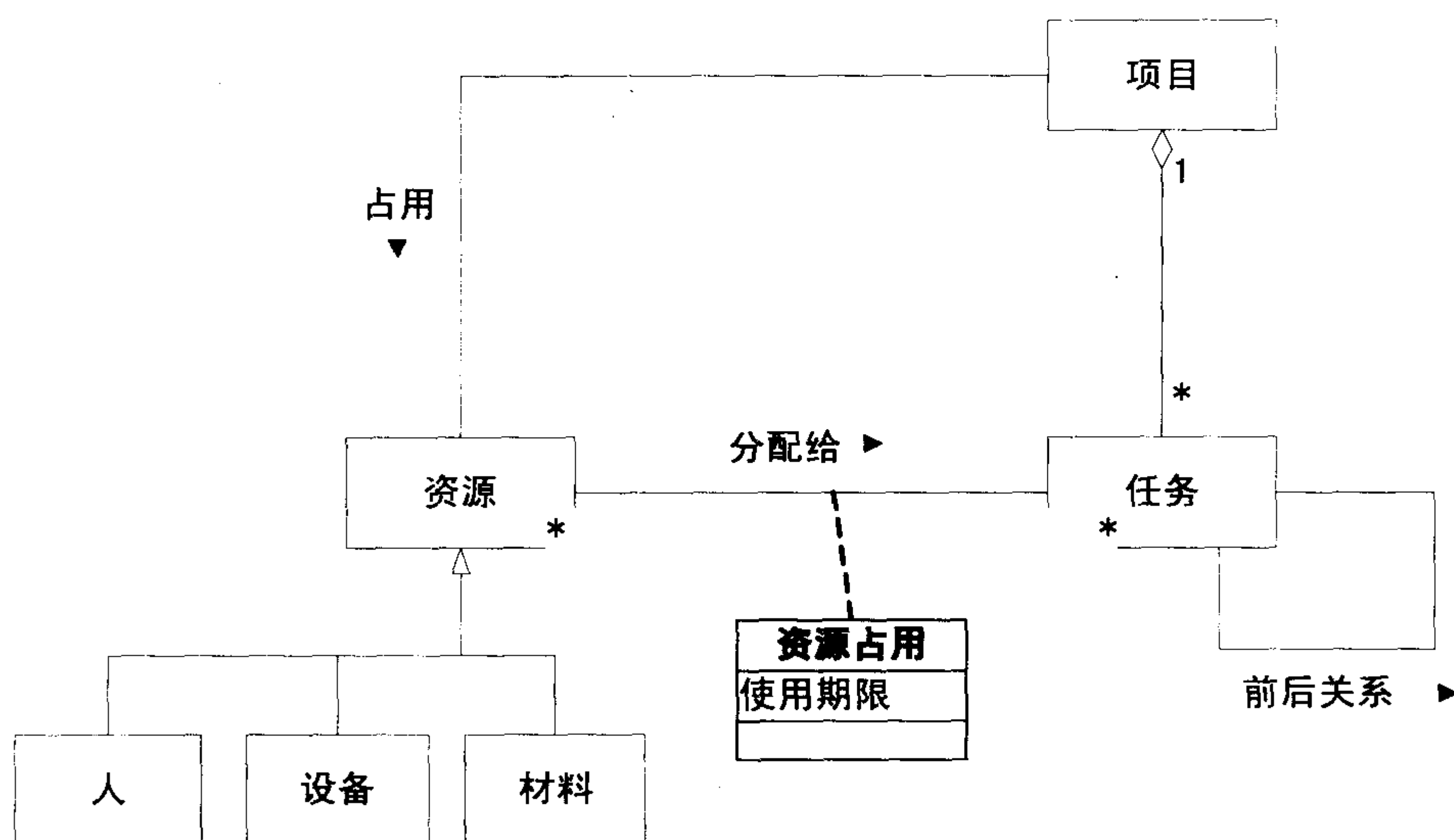


图 12-20 项目、任务、资源的关系越来越清晰了

临近下班的时候，架构师看着已经逐渐有模有样的领域模型（如图 12-21 所示），有一种不大不小的成就感。“我们的模型明天还要继续精化，比如，项目状态的跟踪等问题还没有讨论清楚。”架构师说。

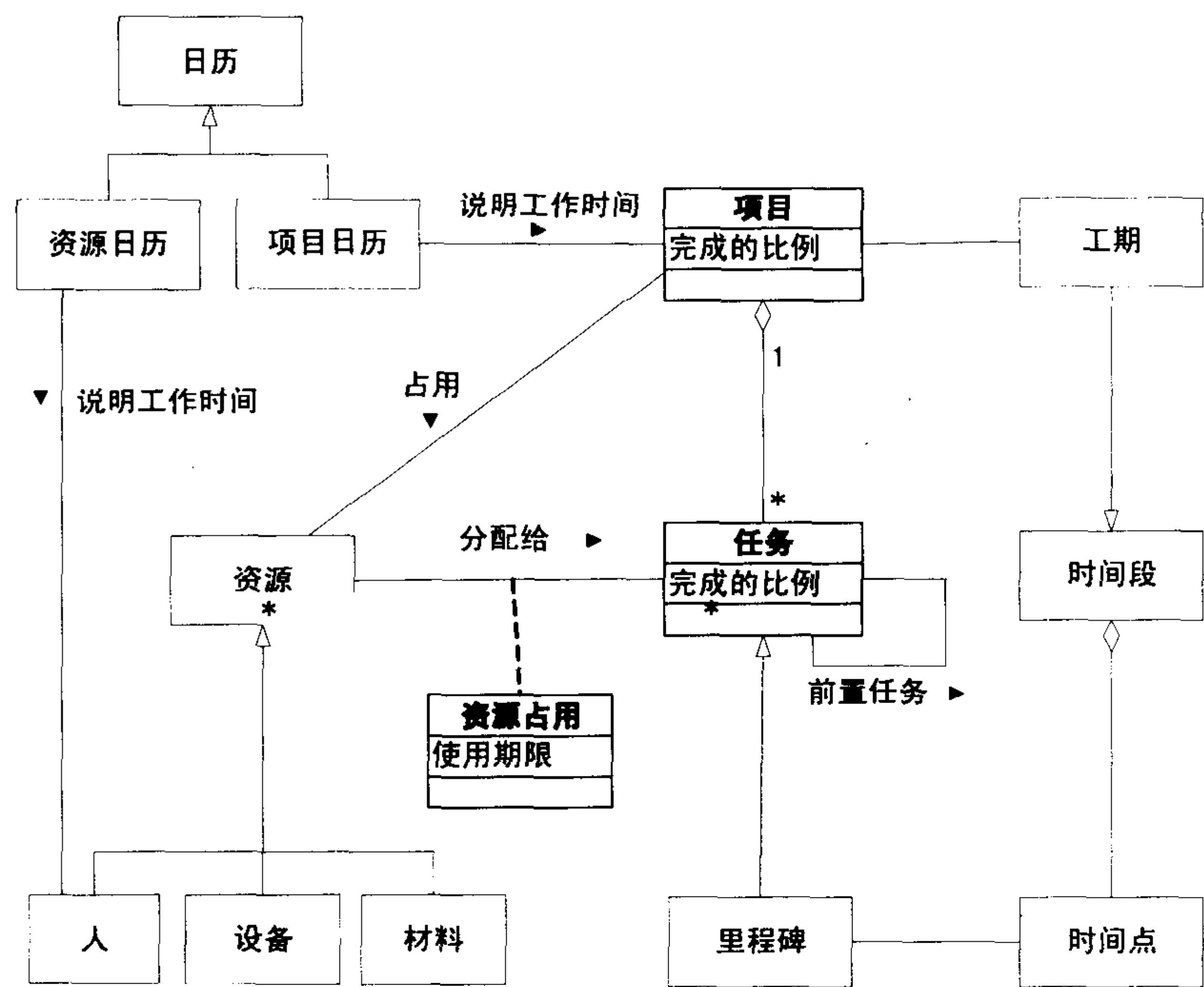


图 12-21 领域建模的阶段性成果

12.5.2 领域建模实录 (2)

第二天，简单回顾了前一天的建模成果之后，新的讨论开始了……

领域专家：PM Tool 应当能够跟踪每个项目的状态，这些状态包括未启动、进行中、已完成、已取消。

架构师：……

领域专家：……

架构师：……（得到了图 12-22）

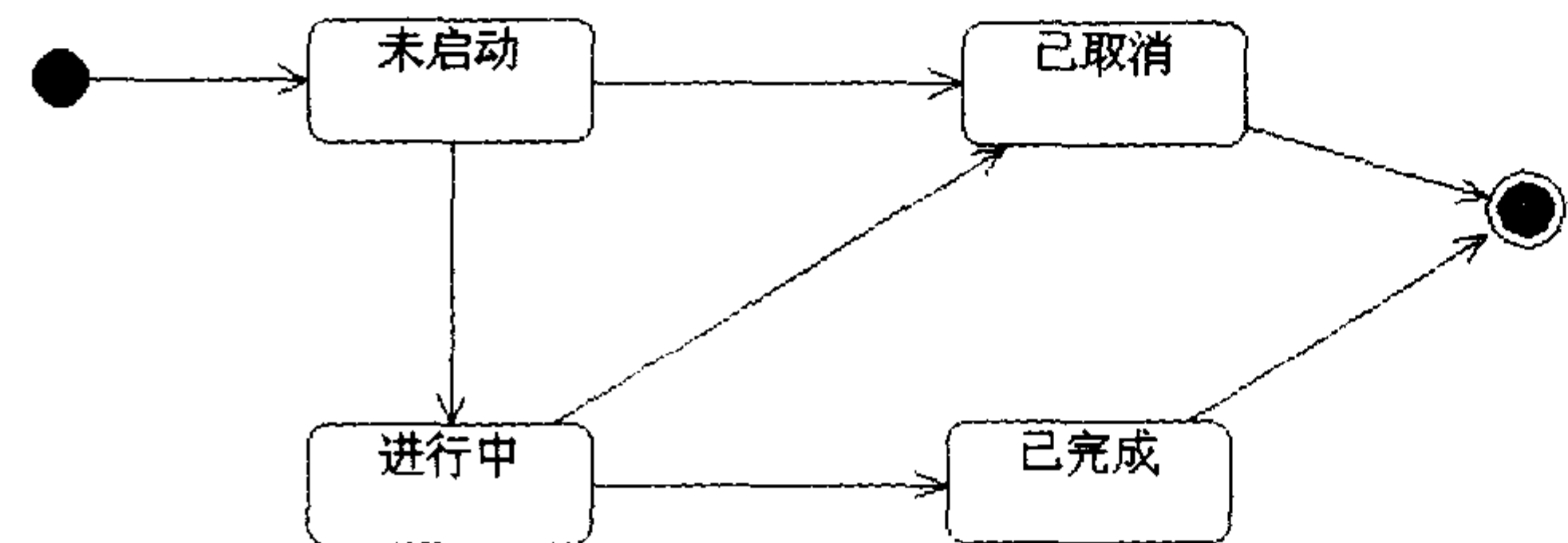


图 12-22 项目可能的状态

领域专家：嗯，很好的图。

架构师：但我想还没完，我们必须明确一个项目的不同状态是因何而发生变化的。

领域专家：嗯，情况是这样的……

架构师：……（得到了图 12-23）

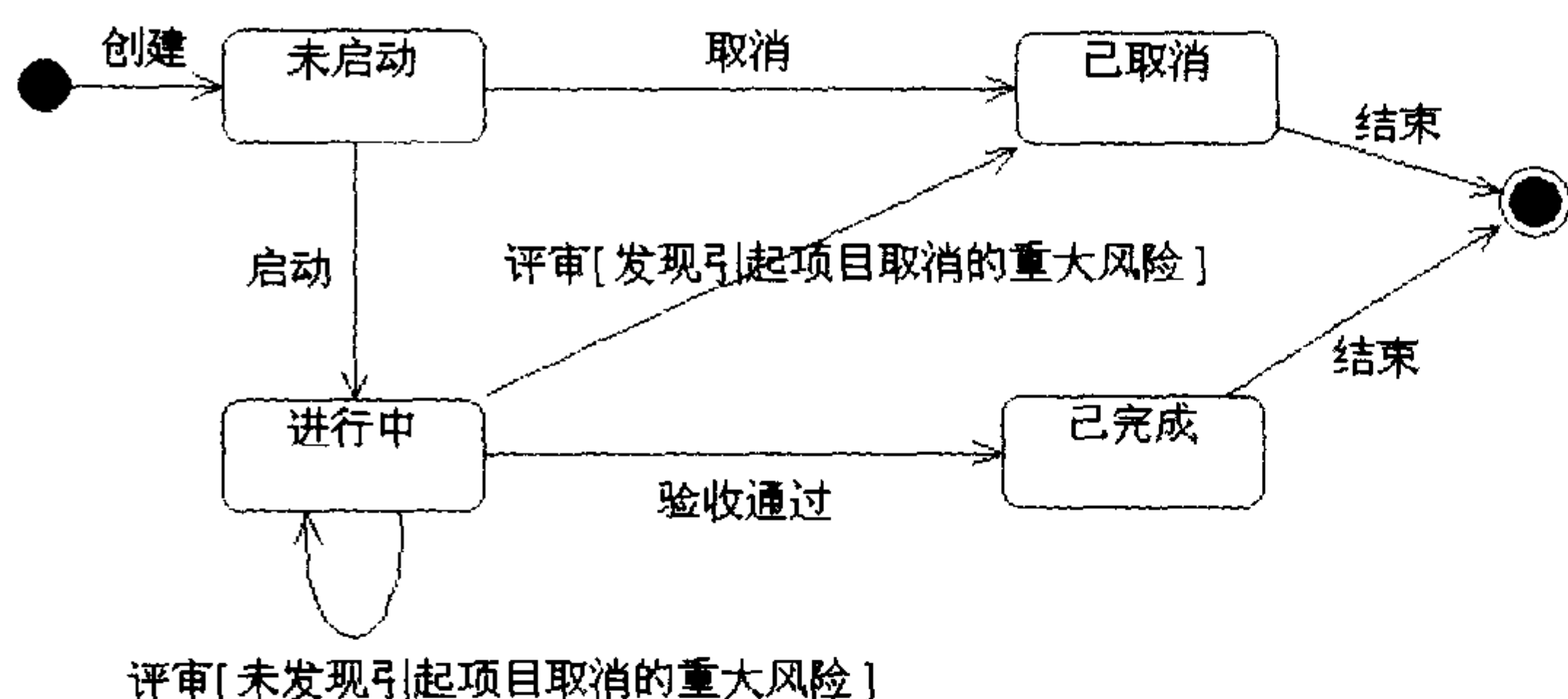


图 12-23 细化了项目转换变化的原因

分析员：项目的每个状态，会对项目团队产生什么影响呢？

领域专家：项目完成和项目取消这样重大的事件，应该通知所有人员知道……

架构师：哦，我知道了，这和你们提到过的公告板功能有关……（得到了图 12-24）

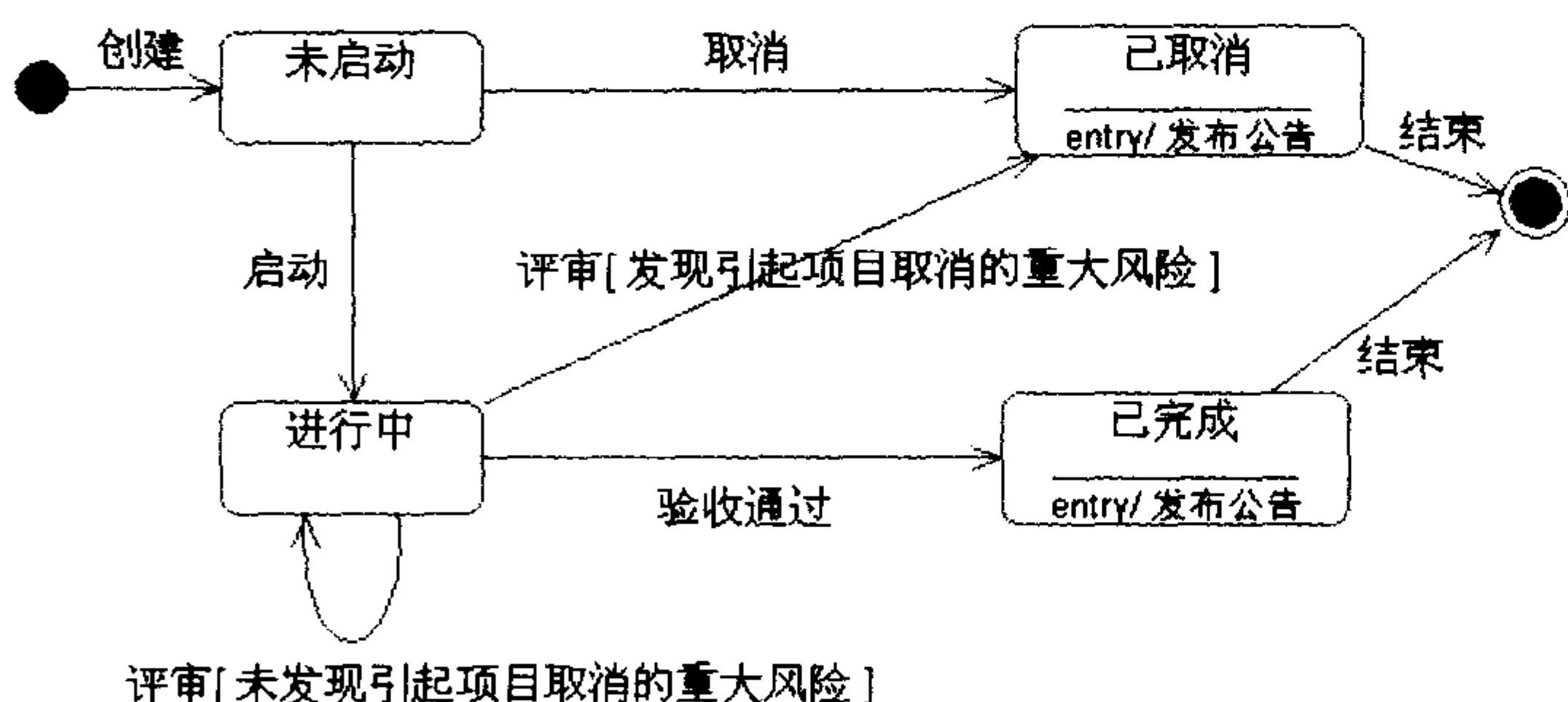


图 12-24 细化了进入状态将引起的业务动作

领域专家：这个系统必须能报告项目的进度状态，包括超前、正常、滞后。

架构师：……

领域专家：……

架构师：很好，我们的领域模型里已经有了时间点、工作量等概念（指着图 12-21 说），通过它们很容易实现计算项目进度状态的算法……（得到了图 12-25）

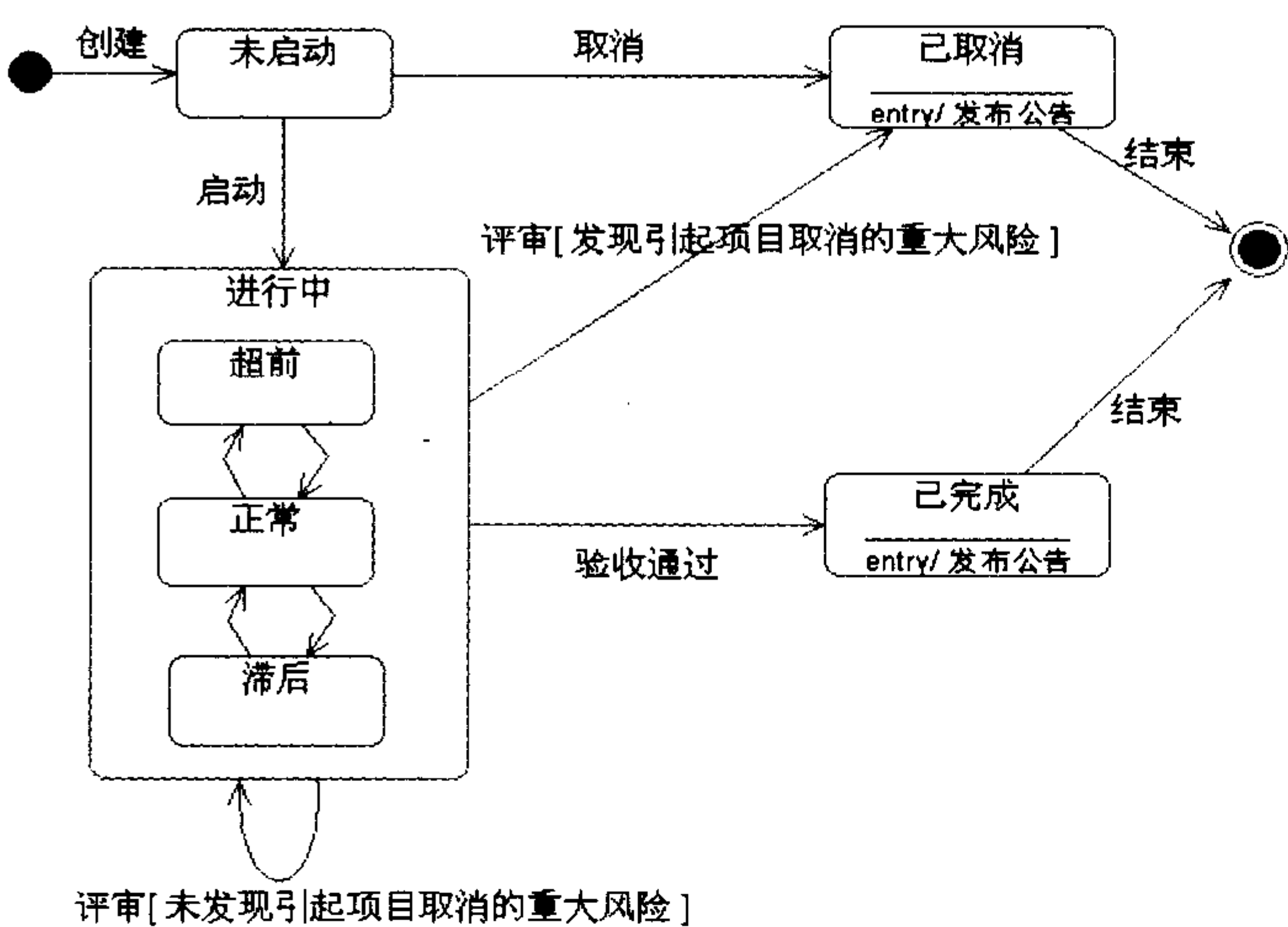


图 12-25 项目的“进行”状态其实包含不同子状态

## 12.6 总结与强调

本章讲述的主题是领域建模。在现代的软件开发中，领域建模已成为大家公认的最佳实践之一。领域模型决定了软件系统功能的范围，并因此而影响着软件系统的可扩展性。

我们中的很多人不重视领域建模。或许，他们并不否认领域模型对于软件系统的重要性；但由于进度紧迫，他们往往抓紧进行“面儿上”的编程事务，而忽视了对软件系统的可扩展性有深远影响的领域模型问题。这印证了《高效能人士的七个习惯》中的那句话，“人人都知道这些事很重要，却因尚未迫在眉睫，反而避重就轻。”



## 第 13 章 确定对软件架构关键的需求

---

关键性的第一步是缩小范围……

——杰拉尔德·温伯格,《你的灯亮着吗》

很少有开发者能奢侈地拥有一个稳定的需求集,或者能够构建一个满足所有已知需求的系统。

——Dean Leffingwell,《软件需求管理:统一方法》

功能、质量和商业需求的某个集合“塑造”了构架。我们把这些塑造需求称为“构架驱动因素”。……知道了构架驱动因素后,就可以开始构架设计了。

——Len Bass,《软件构架实践(第2版)》

关键需求决定架构。简直是醍醐灌顶!

软件架构师没有时间对“所有需求”进行深入分析,这是现实——大多数项目都面临项目工期的压力,软件架构师必须在一定的时间内定夺架构设计方案;否则,没有软件架构所提供的对技术的足够指导以及对分工协作的足够限制,后期的团队开发将面临巨大风险。

软件架构师没有必要对“所有需求”进行深入分析,这是策略——把大部分时间和精力花在对决定架构最重要的一部分需求上,好钢用在刀刃上,最终你设计出的软件架构的质量反而会更高;否则,所有需求的分析都不够深入,导致最终设计出的软件架构可能会流于形式。

### 13.1 虚拟高峰论坛:穷兵黩武还是择战而斗

---

解释一下这两个隐喻。所谓“穷兵黩武”是指把所有需求彻底分析一遍从而设计出软件架构的做法,而“择战而斗”是指为了设计架构仅重点分析对软件架构起关键作用的一部分需求的做法。

读书犹如和作者交谈。本节的写作形式颇为轻松:我们假设把一些高人请到了一起,就“从

软件需求到软件架构”问题展开一个“高峰论坛”（当然是虚拟的）。

### 13.1.1 需求是任何促成设计决策的因素

说到底，一个软件系统的软件架构最终设计成什么样，是由软件需求决定的。

咨询专家 Brian Lawrence 提出：“需求是任何促成设计决策的因素（Anything that drives design choices）。”

需求是任何促成设计  
决策的因素。



### 13.1.2 很少有开发者能奢侈地拥有一个稳定的需求集

“需求决定架构”。话虽这么说，但现实要复杂得多，因为软件需求本身会因需求背景的变化和项目人员的理解等问题发生变更。

正如《软件需求管理：统一方法》的作者 Dean Leffingwell 所说：“很少有开发者能奢侈地拥有一个稳定的需求集……”

很少有开发者能奢侈地  
拥有一个稳定的需求集。



### 13.1.3 关键性的第一步是缩小范围

勿在浮沙筑高台。倘若作为架构设计重要依据的软件需求变化了，你建起的软件架构这个“高台”岂不是要倒塌？

杰拉尔德·温伯格的话让人更深刻地体会到了“运筹帷幄”应有的含义，他说：“关键性的第一步是缩小范围……”

关键性的第一步是缩小范围。



### 13.1.4 要择战而斗

穷兵黩武还是择战而斗，这或许不是问题，因为我们已经倾向于择战而斗了。但问题在于，择战而斗怎么个“择”法。



PeopleSoft 公司的首席技术官 Rick Bergquist 说得精辟：“我的第一个老板 John Grillos 曾说过，要择战而斗。择战的标准如下：它们要具有重要性，它们要具有可能性，它们的数量要少。”

要择战而斗。



### 13.1.5 功能、质量和商业需求的某个集合塑造了构架

方向已经明确了，不是吗？软件架构师要着重深入分析的是软件需求的一个子集，再结合自己的经验，最终设计出软件架构。

卡内基梅隆大学软件工程研究所的 Len Bass 指出：“功能、质量和商业需求的某个集合‘塑造’了构架。我们把这些塑造需求称为‘构架驱动因素’。……知道了构架驱动因素后，就可以开始构架设计了。”

功能、质量和商业需求  
的某个集合塑造了构架。



## 13.2 关键需求决定架构

### 13.2.1 实践中的常见问题

在从需求工作向架构设计工作转移的环节上，我们在实践中暴露出一些问题。对于软件架构师而言，这些问题在一定程度上相当普遍，所以我们一起来解决它们。

**问题一：**抱怨留给架构设计的时间太短，而不是接受项目节奏普遍加快的现实。

从根本上讲，这是没有意识到软件开发所根植于的业务背景——当然，我相信或多或少也受到瀑布模型的影响。无论是对于企业业务还是个人业务，在复杂和高速变化的经济环境里，在对手云集的竞争条件下，软件系统“上马”太慢本身就潜藏着巨大风险。在《非程序员》第 50 期中有一篇来自 Markus Völter 和 Jorn Bettin 的论文《模型驱动软件开发模式》，其中强调新的商业应用的开发期最多不得超过 9 个月：

....

每三个月至少要提供可交付代码。

理想情况下，每三个月应将代码部署到产品中，并得到实际反馈。

新的商业应用的开发，必须在九个月之内“哇哇坠地”，否则就可能危及“妈妈”（开发组）或“婴儿”（应用）的生命……

问题二：认为必须详细分析所有的需求，只有这样才能设计出满足所有需求的软件架构。

有仗就打、有人讨敌骂阵就出战，这种情形在历史小说里经常见到，但往往出现在有勇无谋的武将身上。与此类似，想要将所面对的所有需求都分析一遍的软件架构师是否想过：这是否现实？在有限的时间内，将精力分散在过多的问题上，其结果往往是效果更差。

我们的策略是：关键需求决定架构，其余需求验证架构。

顺着“全面认识需求”的策略思考开去，很容易让人产生疑问：你是在说瀑布式开发吗？当然不是。我们的策略是：在架构设计期间，关键需求决定架构，其余需求验证架构。也就是说，“关键需求决定架构”和“全面认识需求”的策略是不矛盾的。

非关键需求可以用来验证架构，比如以架构方案评审的方式，从每项非关键需求的角度对架构方案进行走查。

问题三：认为软件架构必须是完美的技术解决方案。

关于这一点，Philippe Kruchten 在他的论文《Common Misconceptions about Software Architecture》中明确地进行了批评，并指出架构“够用就好”：

通常，在进行系统架构设计时，时间非常关键。架构师根本没有时间去系统地研究每一种可能的解决方案，以找出最佳解决方案；而是必须快速决策，以便让软件开发工作进行下去。项目开发就像一场“战斗”，如果慢慢吞吞地研究出了最佳解决方案，只怕整场“战斗”早已结束多时了，这显然毫无意义。我经常这样描述架构师的工作：在有些事情并未完全清楚的情况下，快速做出一系列并不算完美的设计决策。架构并不是静态功能，因此无法优化至最佳——无论是设计约束，还是棘手问题，都不会长时间不变而“等”你找到最佳方案。架构不是要完美，而是要足够满意。

( Usually, time is of the essence when designing system architectures. The architects have no latitude to systematically study all possible solution paths and their combinations in order to come up with the optimal solution; they must rapidly make decisions to allow work to proceed. There is no point in coming up with the ideal solution after the battle is lost. I often describe the life of a software architect as a long and rapid succession of suboptimal design decisions taken partly in the dark. It is not a static function that we are optimizing anyway. Neither the constraints nor any parts of the problem are static enough for long enough to approach anything "optimal". Architecture is not about the optimal, or ideal; it is about the adequate, or satisfactory. )



### 13.2.2 关键需求决定架构

采用关键需求决定架构的方法，其好处有二：一曰防守，二曰进攻。

说到“防守”，多少有些“不得不为”的意味。的确如此。

- 一方面，把所有需求统统确定下来之后再开始下游活动是不现实的。需求来自于实践的需要，而实践是发展的，所以“确定的需求”只能是暂时的。于是，我们不得不在“部分需求已确定”的情况下进行架构设计。
- 另一方面，项目何时交付往往是由客户业务的需要决定的，或者是由市场形势决定的，这使得项目工期成了不可改变的“外部限制”（上市时间是一种非功能需求）。时间有限，我们不得不在就项目的业务目标及核心需求达成共识之后开始架构设计，而这时需求并未完全清晰化。

至于“进攻”就是对期望目标的“主动追求”了。我们希望追求的目标有两个：

- 第一，用有限的精力深入分析最为重要的需求。人的思维能力所能应对的复杂性是有限的，因此人们总是有意识地将问题分解、化简和转换。当我们把全部精力放在相对少的需求上时，可以更为深入地分析这些需求，有利于得到透彻的认识，从而设计出合理的架构。
- 第二，因为需求是“促成设计决策的因素”，因此需求的变更可能会引起架构设计不再适合。因此，我们希望能通过所有需求的一个子集来“驱动”我们的架构决策过程，这样可以减少需求变更对架构设计方案带来的冲击，使架构设计趋于稳定。如图 13-1 所示。

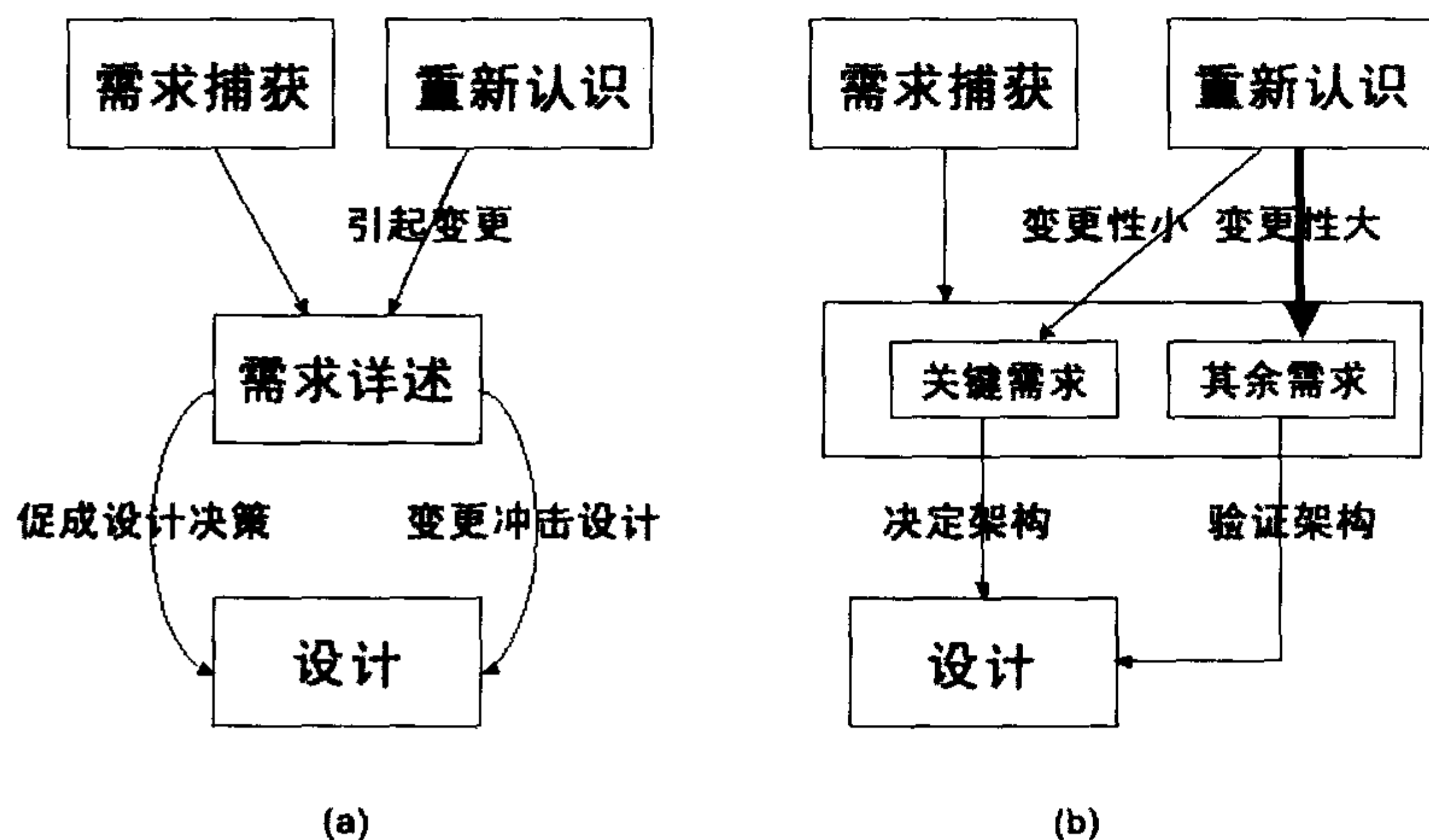


图 13-1 关键需求决定架构，其余需求验证架构

特别是，功能需求的数量是相当巨大的；通过选取不到 20% 的典型功能需求，进行有重点

的深入分析来带动架构设计，可以节约很多时间。如果再考虑到需求变更的问题，在架构设计期间 80%的功能需求的变更都不会对架构设计的推进造成冲击，这太有诱惑力了；毕竟，架构设计之时一般难以达到所有需求都稳定的状态。

### 13.3 确定关键需求在软件过程中所处的位置

#### 13.3.1 对架构关键的需求 vs.需求优先级

在软件过程上游的需求分析活动中，我们有必要识别并记录需求的优先级，这对项目管理乃至控制交付范围等都有重要影响。那么，项目经理所关心的需求优先级和软件架构师所关心的对架构关键的需求之间，到底是什么关系呢？

一“图”以蔽之。如图 13-2 所示，高优先级的需求和对软件架构关键的需求之间，既有区别又有联系。

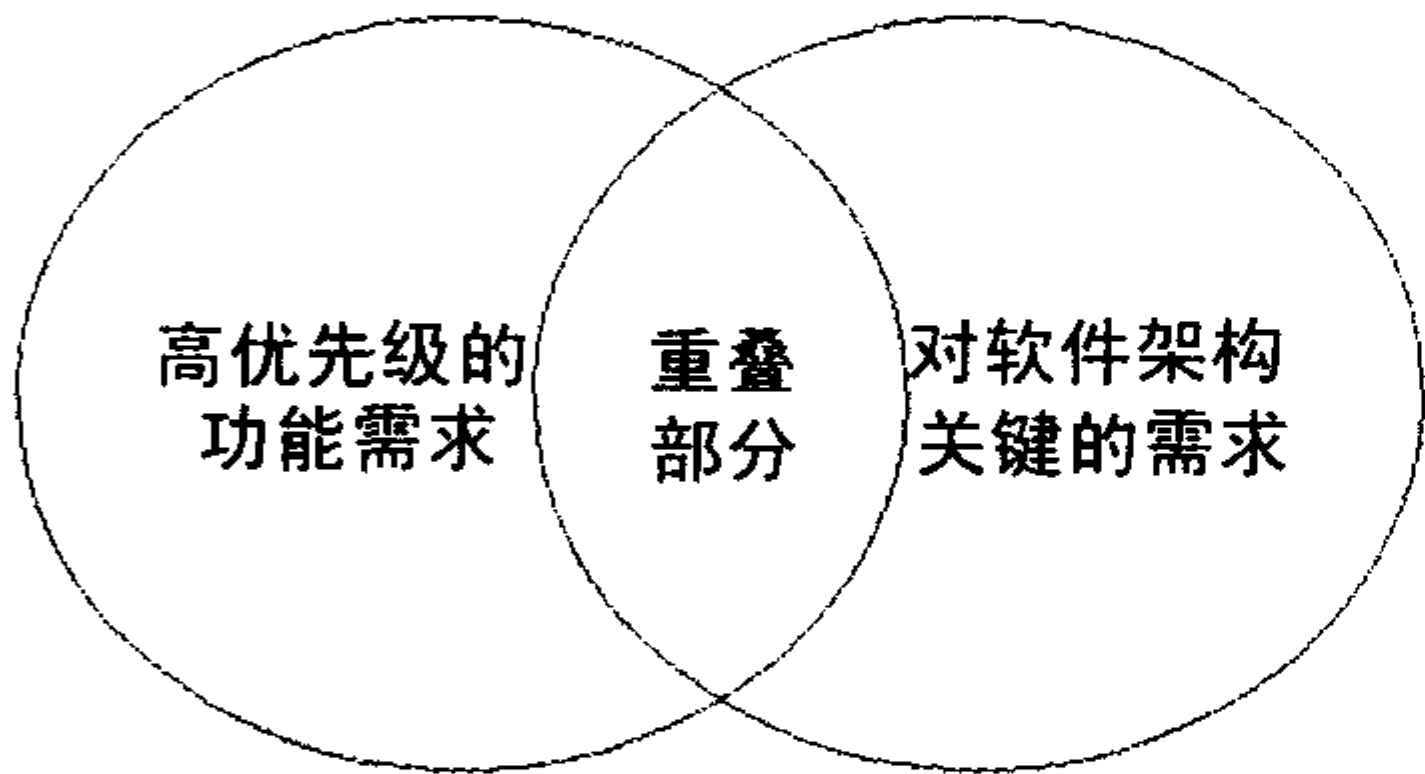


图 13-2 对架构关键的需求 vs.需求优先级

一个事物是否关键、是否优先考虑，要视具体目标不同而定。我们通常所说的“需求优先级”是针对客户而言的（同时要从技术角度考虑需求之间的依赖关系），而本章的主题“对软件架构关键的需求”是对架构设计的影响而言的，请读者注意区分。

需求优先级主要是针对功能需求而言的，除却被依赖的需求应当优先实现之外，需求优先级主要反映了客户希望最终系统提供某功能需求的迫切程度。一般而言，需求优先级可以分为三级：

- 高优先级。必不可少的功能。只有在这些需求上达成一致意见，软件才可能被接受。这些功能的实现质量必须趋于完美；
- 中优先级。必要的功能。这些功能是系统所需要的，如果有必要可以延迟实现。虽然不提供这些功能系统也有可能被接受，但最好不要忽略它们。值得为这些功能的质量付出努力；
- 低优先级。锦上添花式的功能增强。低优先级的需求可以实现也可以不实现；但如果



资源允许的话，实现这些需求将会使产品更臻完美。另外，对于这些需求的实现质量要求不是很高，甚至可以容忍不严重的缺陷存在。

鉴于此，我们也就不难理解，一个项目中，需求优先级为高、中、低的需求的比例应该科学（比如 3:4:3），从而有利于项目管理。如果将需求优先级统统定为高，或者需求优先级为高的需求明显占了压倒性的比例，这显然是不科学的做法，违背了设定需求优先级的初衷，不利于项目管理中权衡与调整。鉴于项目管理不是本书的主题，对此我们不再展开讨论。

总之，可以这么说，需求优先级是项目经理的一种权衡和决策的工具（如图 13-3 所示）。该工具使项目经理可以在一定程度上获得对项目管理的灵活调整的余地，为了在项目的时间、成本和质量要求范围内顺利完成目标，对项目所提供的功能范围（Scope）进行调整。

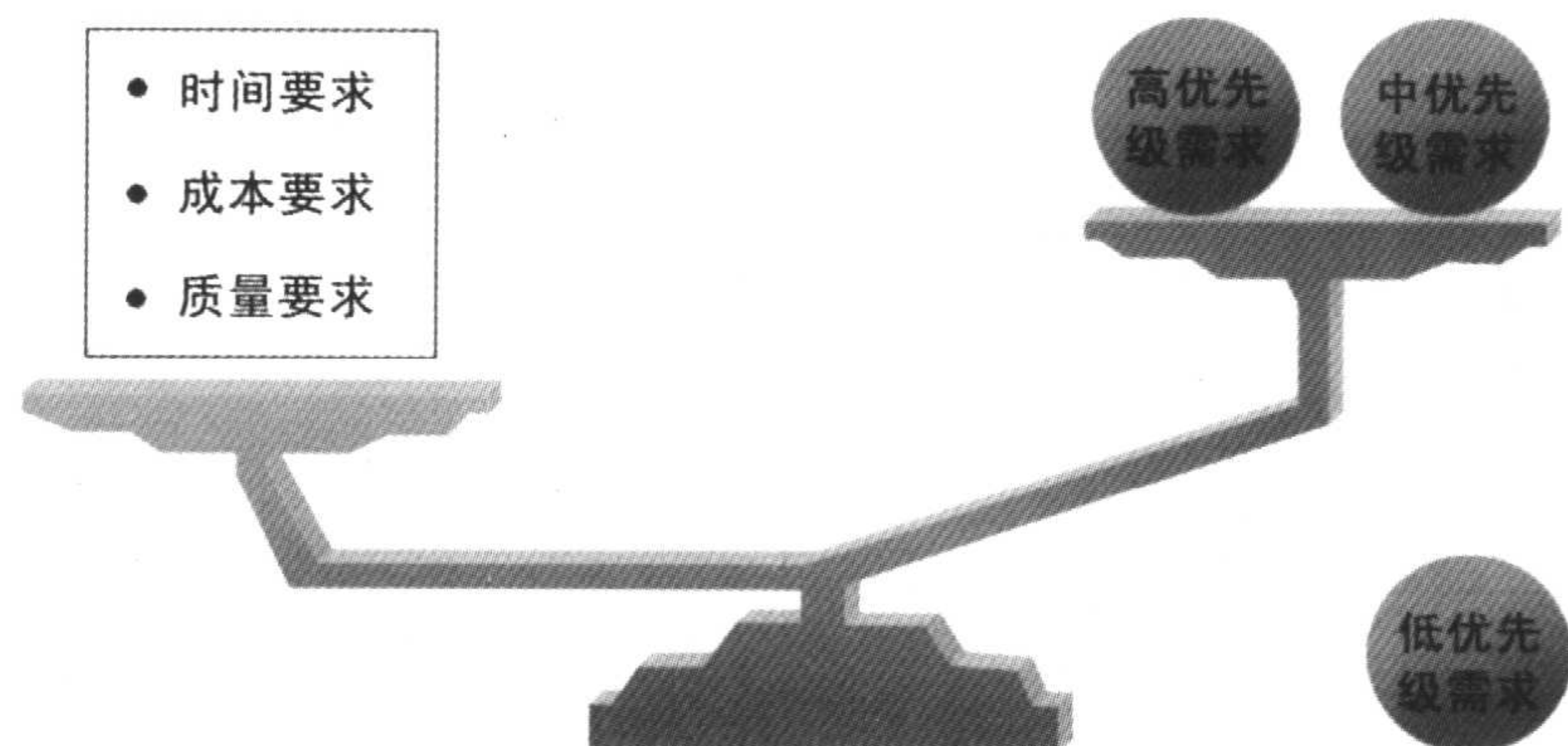


图 13-3 需求优先级是项目经理的一种权衡和决策工具

### 13.3.2 关键需求对后续活动的影响

确定了对架构关键的需求之后，软件架构师下面的活动将主要针对这些关键需求展开（如图 13-4 所示）：

- 无论对于概念性架构的设计（第 14 章），还是实际架构的设计（第 16 章），都需要对关键用例进行分析。此时将采用鲁棒图等用例分析技术，最终将各鲁棒图进行综合，得到整体的软件系统职责划分模型（也被称为逻辑设计模型或分析模型）；
- 质量属性分析中，采用“质量属性—场景—决策”表等技术手段，分析质量属性要求，制定架构级设计决策；
- 当然，经过质量属性分析之后得到的架构决策，可能引起软件系统职责划分模型的调整。以高性能设计为例，无论是“对消息采用多线程并发处理”还是“将图片服务器独立出来以便进行专门的缓存和索引等加速处理”都需要对软件系统职责划分模型进行细化等调整。

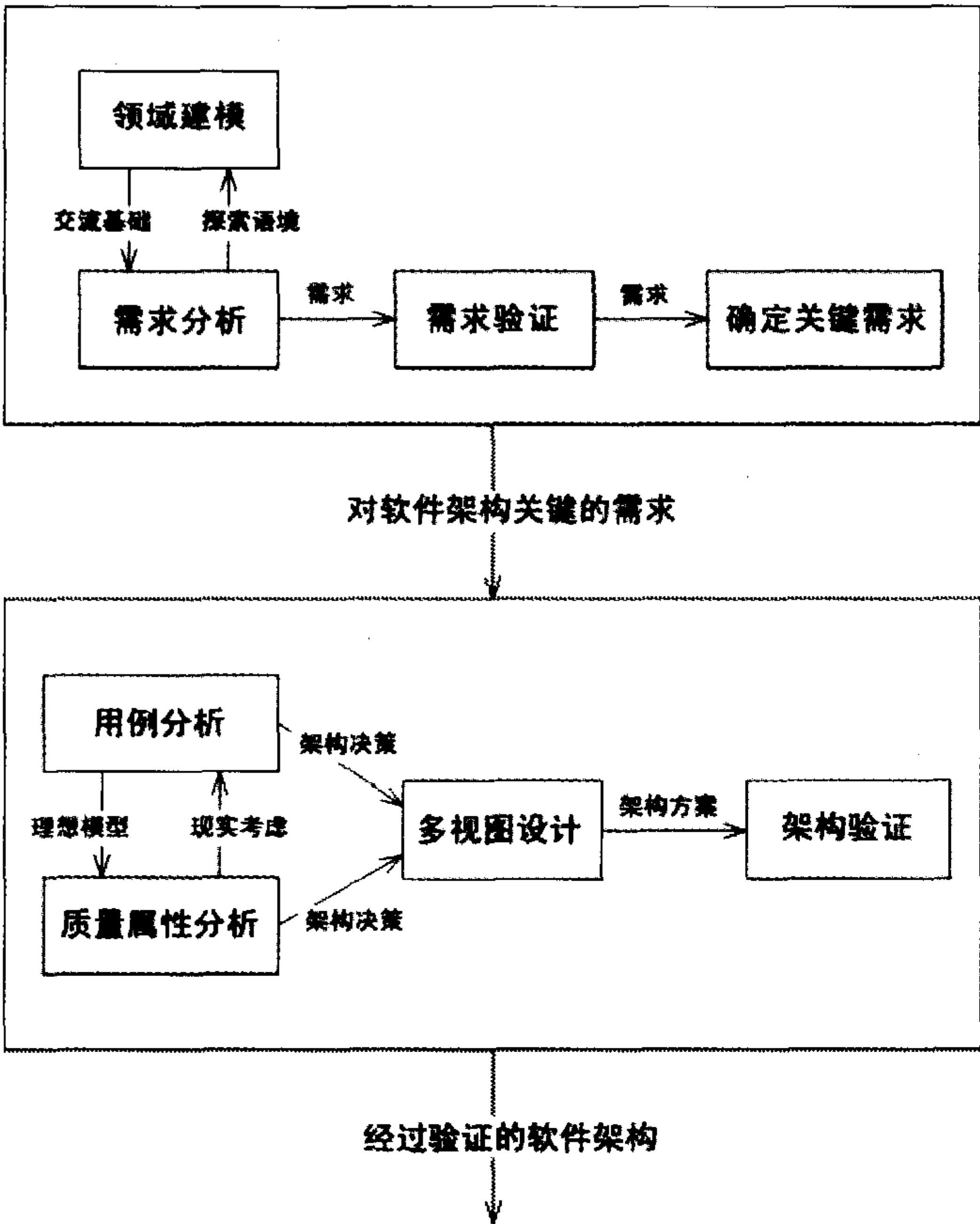


图 13-4 关键需求与后续活动

## 13.4 什么是对软件架构关键的需求

对软件架构关键的需求包括功能需求、质量（属性）需求、商业需求三类，下面一一讨论之。

### 13.4.1 关键的功能需求

任何功能需求，都是由一条特定的“模块协作链”完成的。

所谓软件架构就是关于如何构建软件的一些最重要的设计决策，这些决策往往是围绕将系统



分为哪些部分、各部分之间如何交互展开的。而一个完整的软件功能的实现，则需要这些不同的“部分”之间相互配合，形成一条“模块协作链”，从而才能满足一个完整的应用功能。控制权在模块协作链中来回传递，而功能需求就相当于串起不同模块的线索（如图 13-5 所示）。

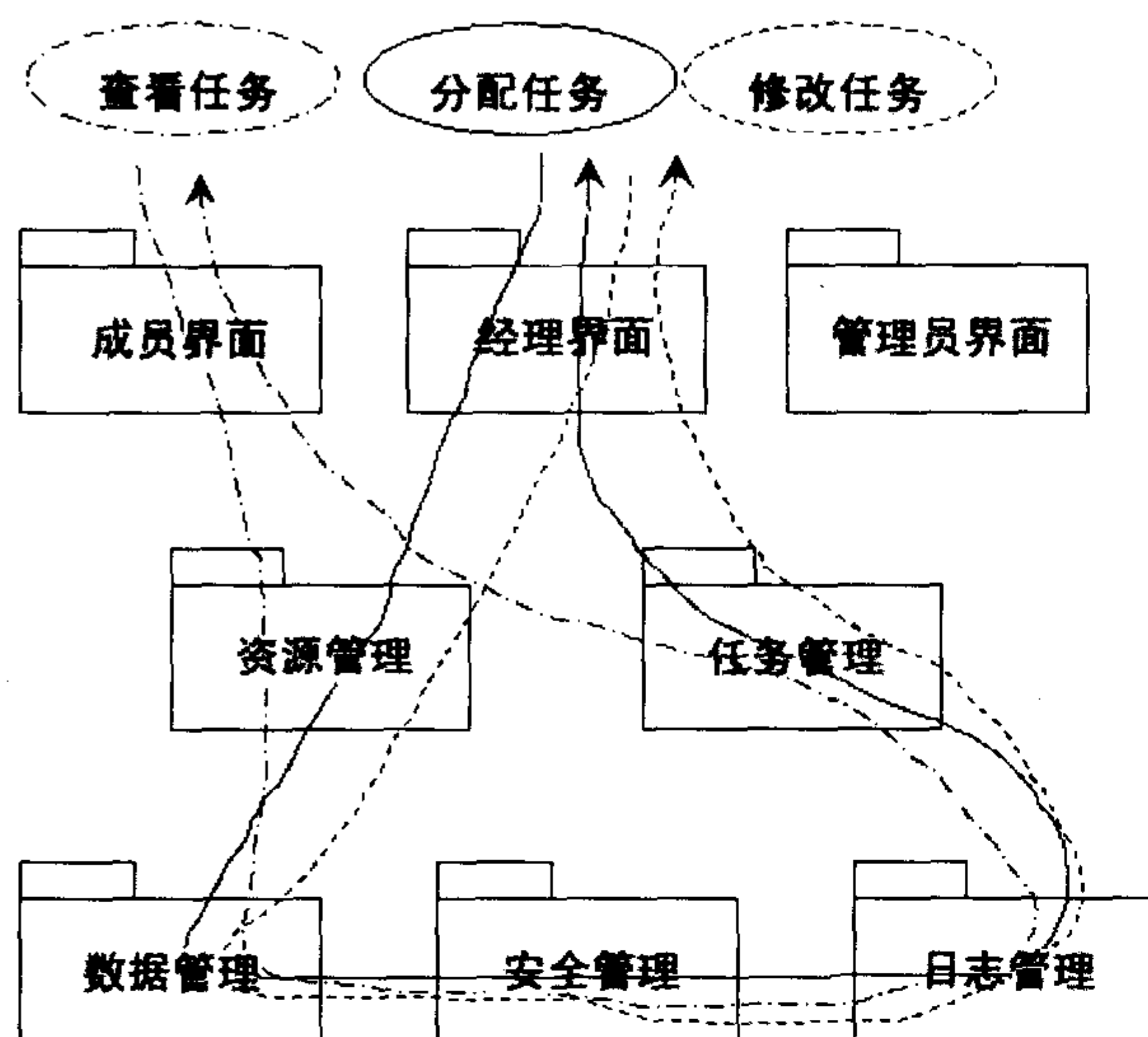


图 13-5 功能需求与模块协作链（参考《AOSD 中文版》）

所以，所谓对软件架构关键的功能需求，就是它涉及（或串起）的模块最多、最典型的功能需求。毕竟，由于功能需求数量众多，其实会有很多功能需求的完成所涉及的“模块协作链”都非常相似。软件架构师通过分析少数关键的功能需求，往往就可以完成一般性的模块划分、职责分配、协作机制定义等和功能需求密切相关的架构设计工作。

### 13.4.2 关键的质量属性需求

要达到高质量属性要求，是有成本的。例如，持续可用性的成本最为明显，它往往要求通过硬件及网络连接的冗余来实现，使成本投入非常可观。因此，现实中一般应慎重考虑对哪些质量属性提出高要求。

另一方面，不同质量属性之间往往具有相互制约性，使得有些质量属性需求同时达到高要求比较困难。例如，“互操作性”需求往往给“安全性”需求造成威胁，而为了满足“高性能”需求，往往需要使用特定平台的非标准能力，这势必影响了系统的“可移植性”，……，不一而足。于是，我们自然应该权衡哪一部分质量属性是架构设计的重点目标。

综上所述，对架构至关重要的质量属性需求是那些经过权衡取舍、最终决定重点支持的质量属性需求。

13.4.3 关键的商业需求

商业需求又称业务需求（其实对应的英文都为 Business Requirement）。一般情况下，商业需求指“组织或客户高层次的目标”。但作为“架构设计驱动因素”的商业需求有着更宽泛的含义：它关注从客户群、企业现状、未来发展、预算、立项、开发、运营、维护在内的整个软件生命周期涉及的商业因素，包括了商业层面的目标、期望和限制等。

目标和期望的例子很多。例如投资开发一个软件系统的企业可能期望“业务处理能力提高 50%”，产品型的软件公司可能期望“半年内该产品的市场占有率达 40%”或者“半年内销售 20 万套”，等等。

出于商业方面考虑的限制因素五花八门，但它们往往对架构设计有很大影响。表 13-1 进行了归纳总结。

表 13-1 商业需求中可能的限制因素

因素分类	因素	对架构的影响
经济因素	成本收益	<div><div>· 预算多少会影响架构师对技术的选择</div><div>· 可重用性、可维护性、可移植性</div><div>· .....</div></div>
	上市时间	<div><div>· 重用程度、技术选型</div><div>· 通过采购模块或平台减少开发工作量</div><div>· .....</div></div>
客户群	多国语言	<div><div>· 架构对多国语言的支持</div><div>· .....</div></div>
	移动与便携	<div><div>· 支持哪些协议、哪些客户端</div><div>· 是否按产品线进行规划</div><div>· 可移植性</div><div>· .....</div></div>
企业现状	遗留系统的集成	<div><div>· 互操作性</div><div>· .....</div></div>
	企业人员和部门分布	<div><div>· 分布式系统架构</div><div>· 可维护性、安全性</div><div>· .....</div></div>
未来发展	期望的系统生存期	<div><div>· 可伸缩性、可扩展性、可移植性</div><div>· .....</div></div>

续表

因素分类	因素	对架构的影响
	阶段性计划	<ul style="list-style-type: none"><li>· 可重用性</li><li>· 可伸缩性、可扩展性、可移植性</li><li>· .....</li></ul>
其他因素	法律法规	<ul style="list-style-type: none"><li>· 可扩展性</li><li>· 可修改性、可维护性</li><li>· .....</li></ul>
	竞争对手	<ul style="list-style-type: none"><li>· 技术选择</li><li>· 易用性</li><li>· .....</li></ul>

## 13.5 如何确定对软件架构关键的需求

图 13-6 展示了确定对软件架构关键需求的步骤，下面我们将一一讨论。

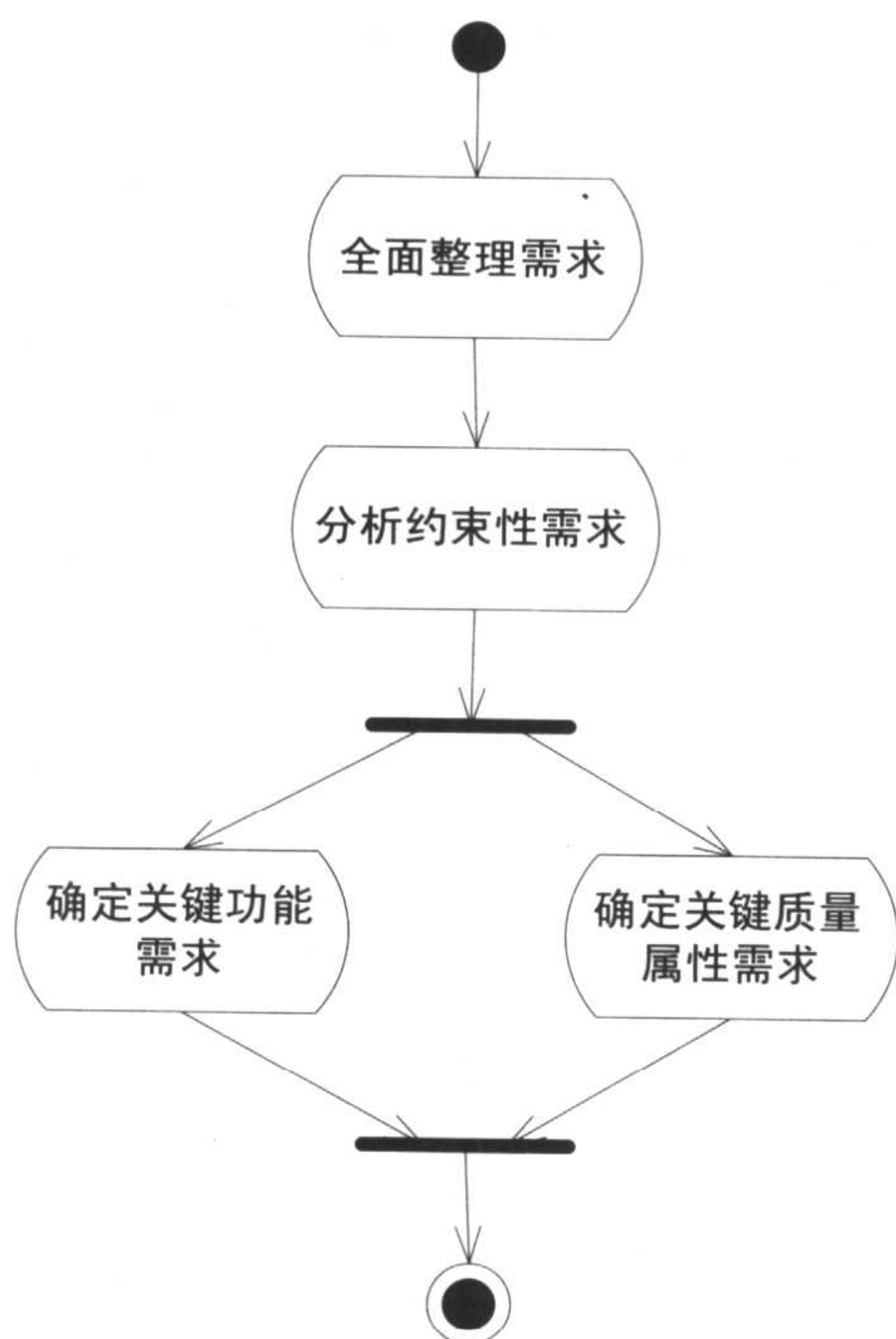


图 13-6 确定对软件架构关键需求的步骤



### 13.5.1 全面整理需求

软件架构师为了确定关键需求，他需要全面整理需求，从而对需求建立通盘认识。为此，软件架构师可能需要：

- 研究《愿景和范围文档》
- 研究《软件需求规格说明书》
- 参加需求讨论会
- 询问客户、用户、领域专家、系统分析员

大多数情况下需求文档未必有软件架构师需要的所有信息，例如易扩展性、易测试性等开发期质量属性往往是《软件需求规格说明书》相对薄弱的环节，所以软件架构师必须通过通盘理解需求，将缺失的、隐含的需求找出来。

如果条件允许，软件架构师应该多参与需求活动，这样更有利于把握需求的轻重缓急。

### 13.5.2 分析约束性需求

对约束性需求进行分析非常有必要，因为很多约束之中“藏”着一些隐含的需求，我们必须将它们找出来。

总体而言，约束性需求可能通过三种途径影响设计（如图 13-7 所示）。

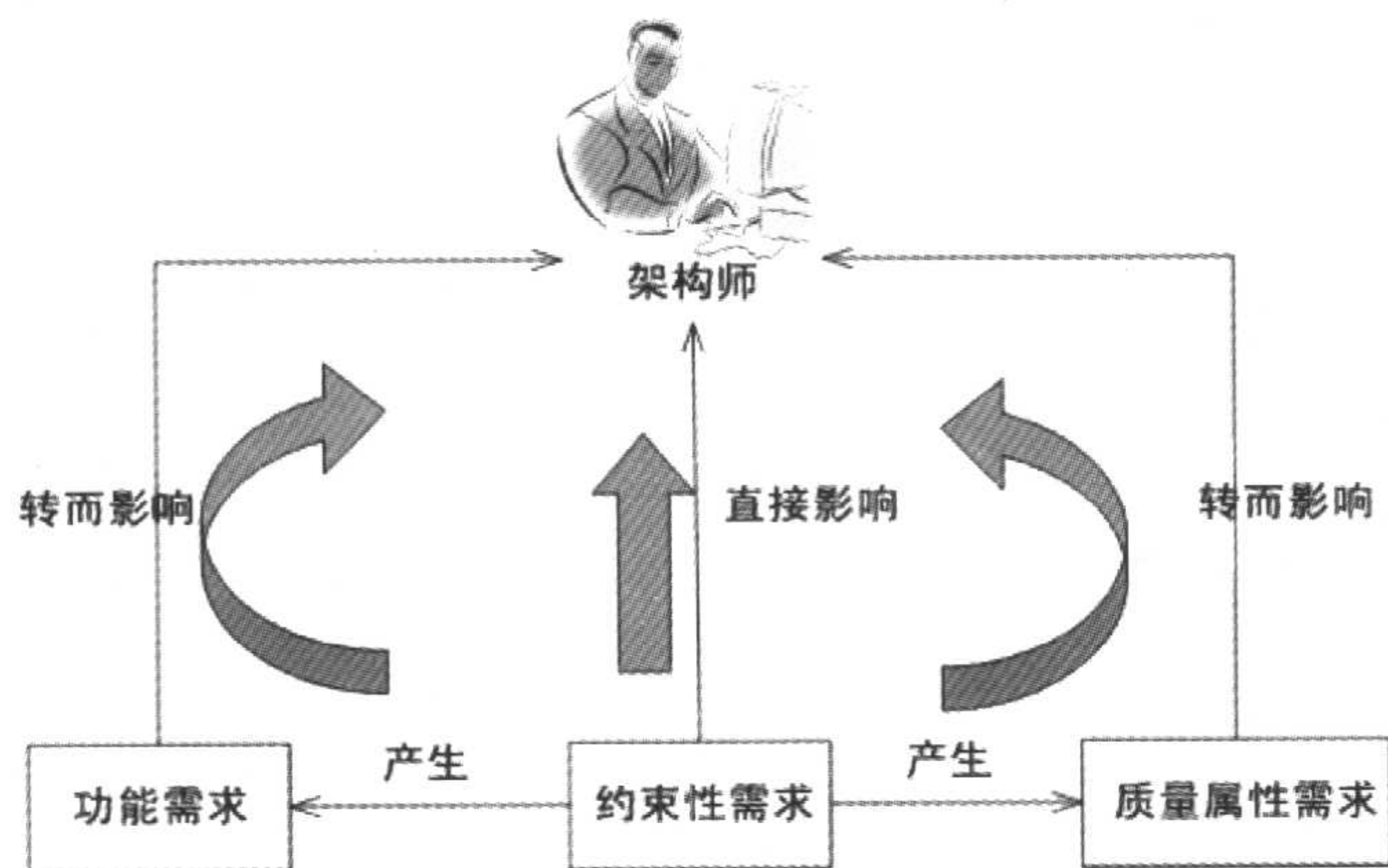


图 13-7 约束性需求影响设计的三种途径

- 直接制约设计决策。例如，“系统运行于 Linux 平台之上”作为一条约束，架构师直接遵守即可；
- 转化为功能需求。例如，“本银行系统必须严格执行人行统一规定的利率”是一条约束，但分析后发现，正是它引出了一条功能需求：即必须提供调整利率设置的实用功能；



- 转化为质量属性需求。例如，有经验的系统分析员发现了一条重要约束：要开发的软件系统的预期用户计算机水平普遍不高。由此，未来的软件系统必须具有很高的易用性（否则不会用）和鲁棒性（否则可能把系统搞瘫痪了）就是非常必要的啦。

### 13.5.3 确定关键功能需求

如何确定关键功能需求？

对此，Per Kroll 和 Philippe Kruchten 在《Rational 统一过程实践者指南》中所论述的相当令人信服（或许读者需要一点 RUP 知识基础）：

在初始阶段，应该确定出一些（大概占总数的 20% 至 30%）对系统起关键作用的用例。这些用例通常对创建架构具有重要影响。

**A. 作为应用程序的核心或实现了系统的主要接口的功能**，因此，对系统架构有很重要的影响。通常系统架构师通过分析冗余的管理策略、资源互斥风险、性能风险和数据安全风险等来识别出哪些用例是最重要的。例如，在销售系统中，从信用卡划账和支付是最主要的用例，因为它是与信用卡确认系统的接口，它的负载和性能特性也很重要。

**B. 必须被实现的功能**。这些功能反映了系统的本质，如果这些功能不能实现，那么开发出的应用程序就没有价值了。通常领域专家和相关方面的专家知道用户最需要哪些功能（主要行为、数据处理的峰值和关键的控制操作等）。比如，如果没有接受订单的功能，就不能实现一个订单发布系统。

**C. 覆盖了系统架构的一些方面，但没有被其他重要的用例覆盖到的功能**。要确保解决所有主要技术风险，就必须全面了解系统的每个方面。否则，即使架构中的某个方面看起来没有重大风险，但是在设计、实现和测试时就很有可能发现这个部分中潜伏着巨大的技术风险。

读者还要识别用户需求中的一些难于实现的、未知的或者存在风险的元素（也许包括一些非功能性的元素），并且要找到一些用例（或者用例的一部分）来说明当前遇到的困难以及解决方案的风险。在这个过程中，通常会有一些技术性风险转移到系统架构的基础部分中。例如，如果系统对时间响应特性或负载特性有特殊的要求，就要识别出一个用例（或者用例的一个事件流）来说明这个需求，同时还要表现出对特性的要求。还有其他一些例子，比如错误恢复策略和系统初始化策略等。

最后，还要识别出这样的一些用例：它们既不会对系统架构产生重要影响也没有技术风险，但是它们描述了尚未涉及到的系统功能。这样，在细化阶段结束时，才能创建出体现系统要领的整体架构。例如，要确保每个主要的“业务实体”都至少与一个核心用例交互。

### 13.5.4 确定关键质量属性需求

为了确定对架构设计关键的质量属性需求，需要做如下三方面的工作：

- 考虑为了提高要开发的软件系统受认可的程度，应着重提高哪些方面的质量属性要求；
- 接下来，充分考虑这些质量属性的相互制约或相互促进关系，以调整不同质量属性的要求标准——例如，你可能会决定高性能要求最最重要，而可扩展性也比较重要；
- 同时，必须满足各种约束性需求。

## 13.6 PM Tool 实战：确定关键需求

表 13-2 列出了 PM Tool 架构师确定的对软件架构设计最为关键的需求子集。

表 13-2 关键需求

非功能需求			功能需求
约束	运行期质量属性	开发期质量属性	
客户群工作的平台多样化	跨平台运行	可扩展性	创建项目
成本效益考虑	易用性		查看项目信息
应考虑外包趋势	互操作性		添加项目任务
和其他系统交换数据			从 HR 系统导入资源发布通知

不难看出，通过类似这种表格的形式进行思维有利于“全面整理需求”；接下来，必须“分析约束性需求”，把有些遗漏的需求找出来，确保约束真正被满足。例如，“和其他系统交换数据”意味着 PM Tool 应当提供“从 HR 系统导入资源”的功能。

继续下去，应“确定关键功能需求”。例如，为什么“从 HR 系统导入资源”是对架构设计关键的需求呢？这是因为该功能涉及了 PM Tool 与外部系统接口的模块，而其他功能没有“覆盖”这一点，所以架构师应特别关注这个功能的设计。

最后，应“确定关键质量属性需求”。让人高兴的是，例子中清晰地反映了一个架构师必须了解的现实：很多对架构关键的质量属性都和包括商业层面的目标、期望和限制等在内的“商业需求”有直接关系。这也从一个侧面说明了本书建议架构师采用的架构设计方法，和软件架构师所承担的责任（为不同的涉众负责）相匹配的一面。



# 13.7 总结与强调

事情简单勇者胜，事情复杂智者胜。

面对时间压力和“复杂性怪兽”，我们有理由置疑“分析透彻所有需求，然后设计出架构”的做法。回忆我们曾讲述过的“全面认识需求”的必要性，我们思考的焦点自然而然地落到了“需求广度和需求深度”的问题上来。也就是说，“全面认识需求”要求我们要关注需求的广度，特别是要覆盖到质量属性需求和约束条件等非功能需求；但要进行架构设计，必须在对需求整体了然于心之后抓住关键。“关键需求决定架构”策略的哲学就藏在需求广度与需求深度的权衡之间（如图 13-8 所示）。

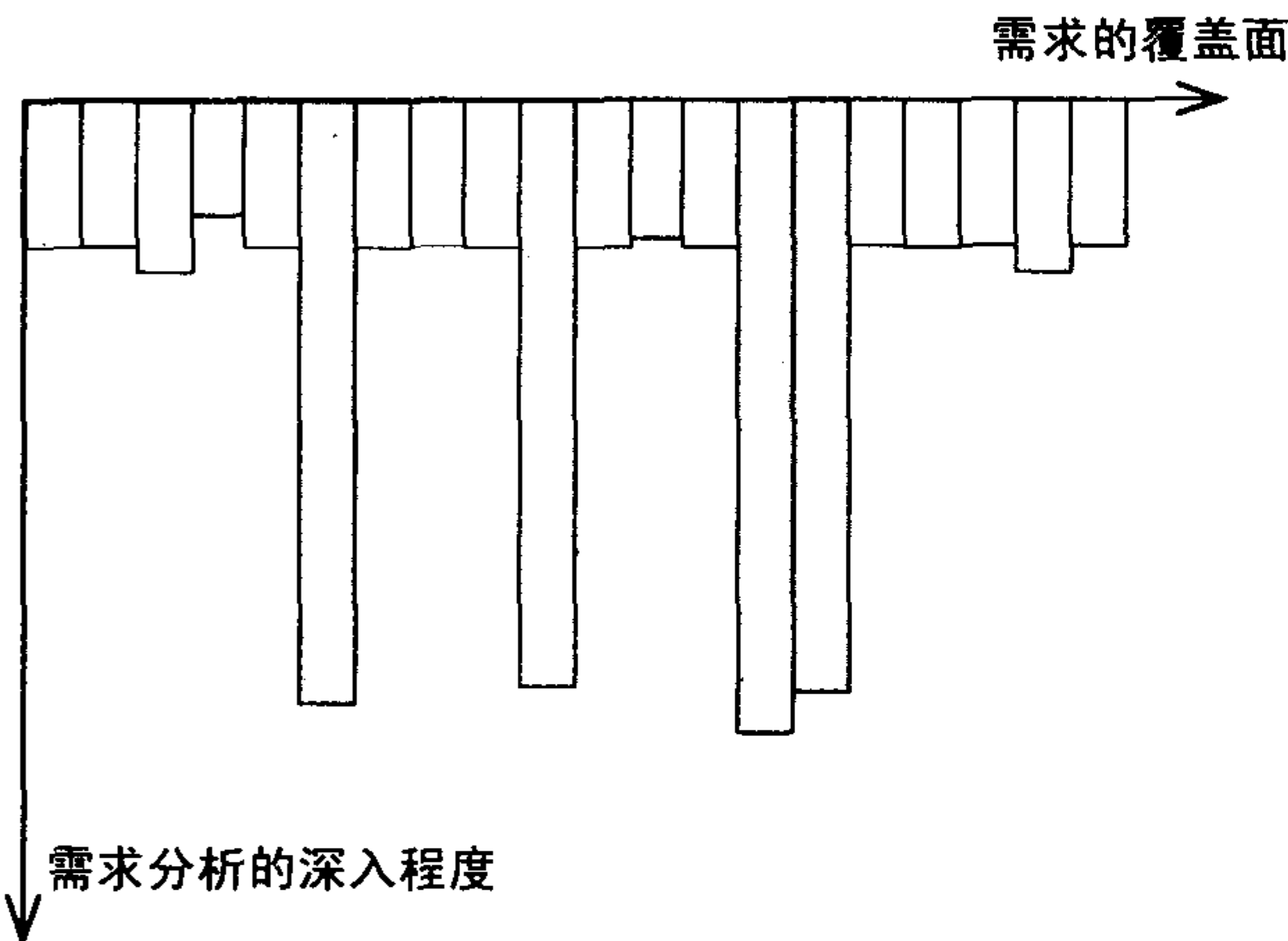


图 13-8 将广度和深度分别对待





## 第 14 章 概念性架构设计

少则得，多则惑。

—— 老子，《道德经》

要表达一件待完成的事情，常常需要对基本元素进行意料不到的复杂组合。

—— Frederick P. Brooks, 《人月神话》

体系结构处理清晰定义和沟通系统结构的原则、机制、模式和结构。在健壮性分析阶段中，要第一次考虑体系结构，并第一次定义高层类型结构，然后再在设计阶段为体系结构增加细节。

—— Ivar Jacobson, 《软件复用：结构、过程和组织》

“功能、质量和商业需求的某个集合塑造了构架。”这真是一句架构箴言！

软件系统的概念性架构设计旨在规划关键问题的解决策略，为此，我们必须考虑关键的功能需求、关键的质量属性要求，以及商业层面的目标和限制。另一方面，架构设计并非易事，因此必须把“经验”真正用起来，例如自身知识、业界成果和架构模式等。图 14-1 归纳了这一“上”一“下”两方面的因素——于是本章的“架构”也就明确了（毫无疑问，书也有架构）。

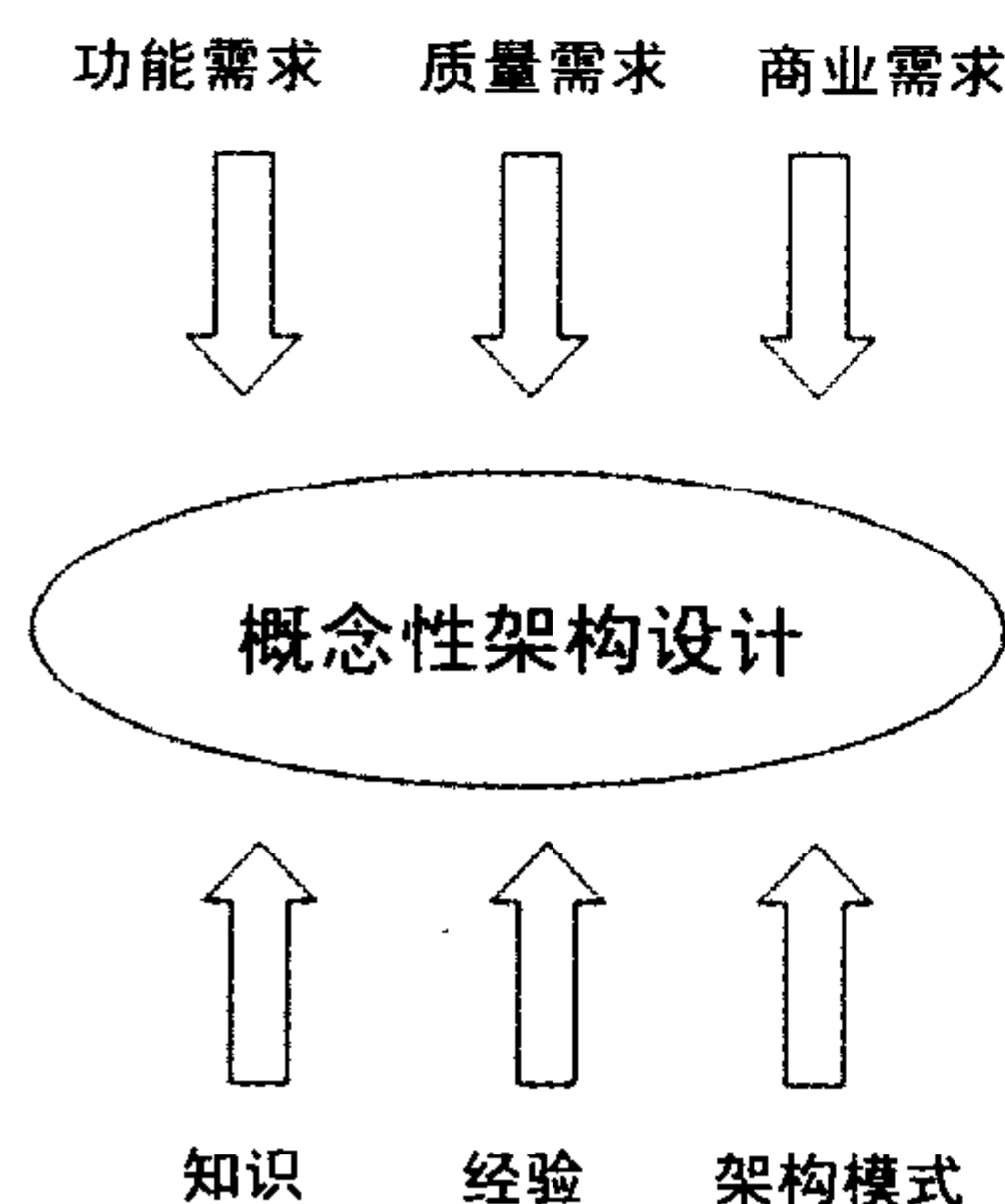


图 14-1 概念性架构设计

## 14.1 概念性架构设计的步骤

概念性架构设计大致可以分为如下三步。

**第一步，鲁棒性分析。**所谓鲁棒性分析是这样一种方法：通过分析用例规约中的事件流，识别出实现用例规定的功能所需的主要对象及其职责，形成以职责模型为主的初步设计。

不难看出，鲁棒性分析是从功能需求向设计方案过渡的第一步，所获得的初步设计是一种理想化的职责模型，它的重点是识别组成软件系统的高级职责块、规划它们之间的关系。这个职责模型是规划架构机制、满足高质量属性要求的武器。

**第二步，引入架构模式。**架构模式也称为架构风格，是软件界长期实践的经验结晶。架构模式的核心是架构机制，即以“架构模式=组件+连接器”的形式明确关键设计元素和关键交互方式。

在实践中，可以将架构模式和架构隐喻等同看待，架构师当然可以创造自己的隐喻，总之明确架构机制是最关键的。

**第三步，质量属性分析。**质量属性需求如此重要，以至于忽略关键质量属性要求往往会导致架构设计的失败。质量属性分析给软件架构师提供了明确的机会去思考和解决这一问题，所以有利于提升架构设计质量。

质量属性分析可以采用“属性—场景—决策”表方法，我们在第 15 章将专门讲述这种方法。

图 14-2 对概念性架构设计的步骤和成果进行了说明。鲁棒性分析产生的职责模型是引入架构模式、进行质量属性分析的基础，并在这一过程中被调整、被细化，也被组织得更有规律。引入架构模式是为了确定关键的交互机制，而质量属性分析将针对质量需求制定相关设计决策。

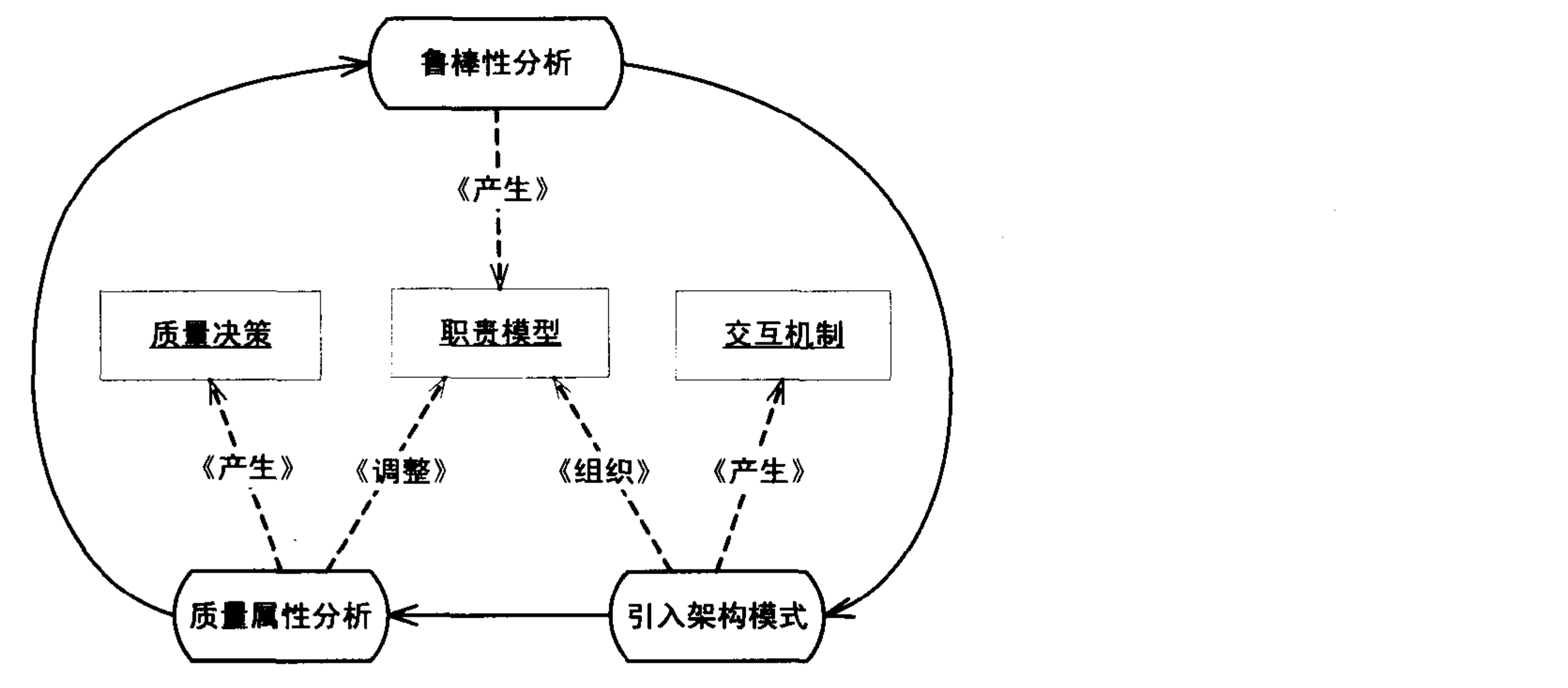


图 14-2 概念性架构设计的步骤



实践中不必过于严格地遵守概念性架构设计的所谓步骤，这是因为解决任何复杂问题的前期都有很大的探索成分，而过于拘泥于步骤反而会限制思维。

## 14.2 鲁棒性分析

### 14.2.1 分析和设计之间的鸿沟

很多人会在需求分析之后卡壳——不知道怎么做。

过渡不好就会卡壳，从需求分析向架构设计过渡时遇到的困难也是如此。鲁棒性分析技术是由 Ivar Jacobson 提出的，它其实是一种设计技术。之所以被冠以“分析”的名字，是由于它仅关注功能需求，根据功能需求导出初步设计，并能“返过头”来帮助发现用例规约中的遗漏和错误之处。

的确，用例规约描述了待开发软件系统的使用方法，却没有以类、包、组件和子系统等元素的形式来描述系统的内部结构。从用例规约向这些设计概念过渡之所以困难，是因为如下一些原因。

- 用例是面向问题域的，设计是面向机器域的，这两个“空间”之间存在映射；
- 用例技术本身不是面向对象的，而设计应该是面向对象的，这是两种不同的思维方式；
- 用例规约采用自然语言描述，而设计采用形式化的模型描述，描述手段也不同。

用 Doug Rosenberg 的话说，这叫“分析与设计之间的缺口”，如图 14-3 所示。

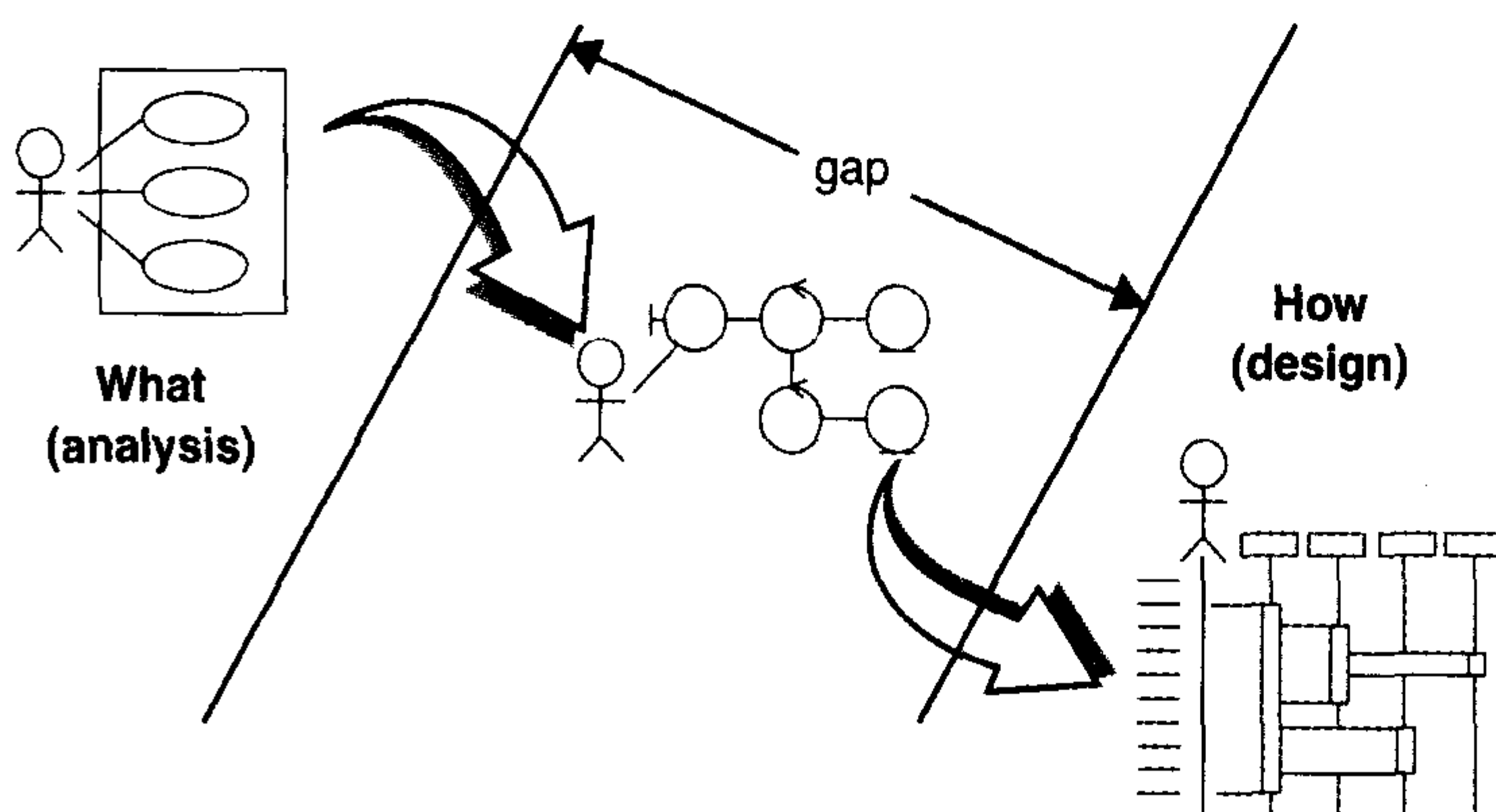


图 14-3 分析与设计之间的缺口（图片来源：《UML 用例驱动对象建模》）

从用例到对象设计的过渡可以采用不同的技术，但鲁棒图是最好的。不难理解，从需求分析到架构设计的过渡可以有多种技术，这些技术既可以像用例规约技术一样具有较强的结构化特

点，也可以像未来的设计模型一样是面向对象的。关于这一点，ICONIX 方法的创始人 Doug Rosenberg 在 2005 年 10 月的《程序员》专访中说：“很多人用活动图来展开用例中的一些细节情况，但是我们采用的是鲁棒图。鲁棒图相比活动图最大的一个好处就是可以让你所关注的设计真正能够把用例转化成为对象（实体对象）以及 GUI（界面对象），相反，活动图则更多地关注如何将 these 设计转化成 if-then-else 这样的逻辑判断，就像流程图一样。”

14.2.2 鲁棒图简介

鲁棒图包含三种元素（如图 14-4 所示），它们分别是边界对象、控制对象和实体对象。



图 14-4 鲁棒图的元素

边界对象对模拟外部环境和未来系统之间的交互进行建模。边界对象负责接收外部输入、处理内部内容的解释，并表达或传递相应的结果。

控制对象对行为进行封装，描述用例中事件流的控制行为。

实体对象对需要存储的信息进行描述，它往往来自领域概念，和领域模型中的对象有良好的对应关系。当然，对面向对象而言实体对象也有一定的行为。

鲁棒图的三种对象很好地概括了实际系统中对象三类职责：交互、控制、信息。并且，令架构师高兴的是，这三种职责和组成架构的抽象元素有完美的对应关系：连接元素、处理元素、数据元素。很多人熟悉 MVC 架构，可以认为边界对象、控制对象、实体对象分别对应视图、控制器、模型。如果考虑到面向对象中“良好的对象”应该是数据和行为的封装体，则可以认为实体对象（和数据元素）仅是 MVC 中“模型”的一部分，如图 14-5 所示。

连接元素	边界对象	视图
处理元素	控制对象	控制器
		模型
数据元素	实体对象	

图 14-5 三种分类之间的对应关系

图 14-6 展示了鲁棒图的建模规则。Doug Rosenberg 在《UML 用例驱动对象建模》中写道：

通过以下 4 条语句，可以理解该图的本质：

- 1. 参与者只能与边界对象交谈。

2. 边界对象只能与控制体和参与者交谈。
3. 实体对象也只能与控制体交谈。
4. 控制体既能与边界对象交谈，也能与控制体交谈，但不能与参与者交谈。

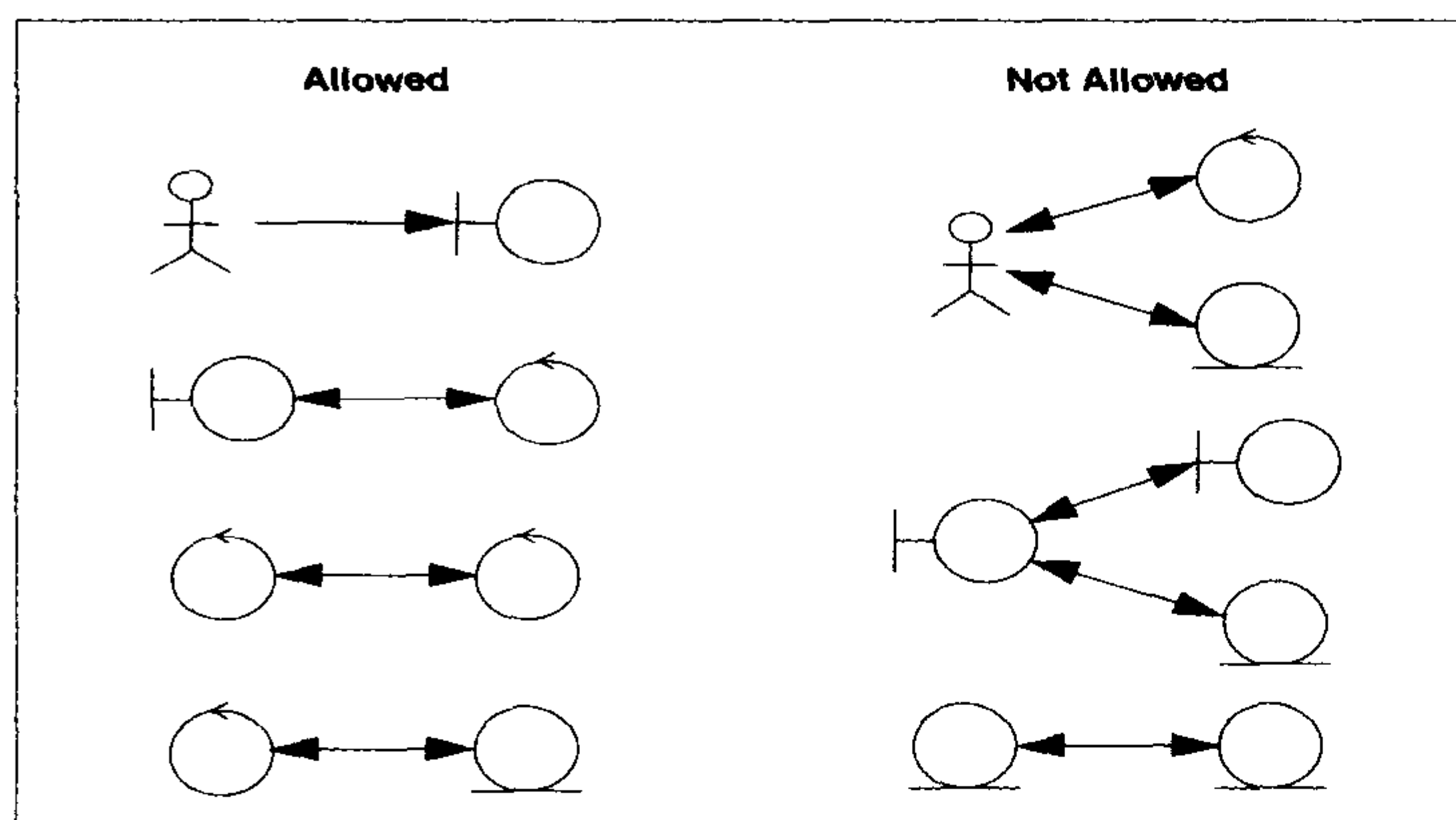


图 14-6 鲁棒图建模规则（图片来源：《UML 用例驱动对象建模》）

### 14.2.3 从用例到鲁棒图

要建立概念性架构，应明确实现预期功能所需的职责模型，研究用例执行的不同场景是个不错的办法。就面向对象系统而言，一个场景的“产生”是一连串的对象协作的“结果”，通过分析场景可以发现这些对象，如图 14-7 所示。

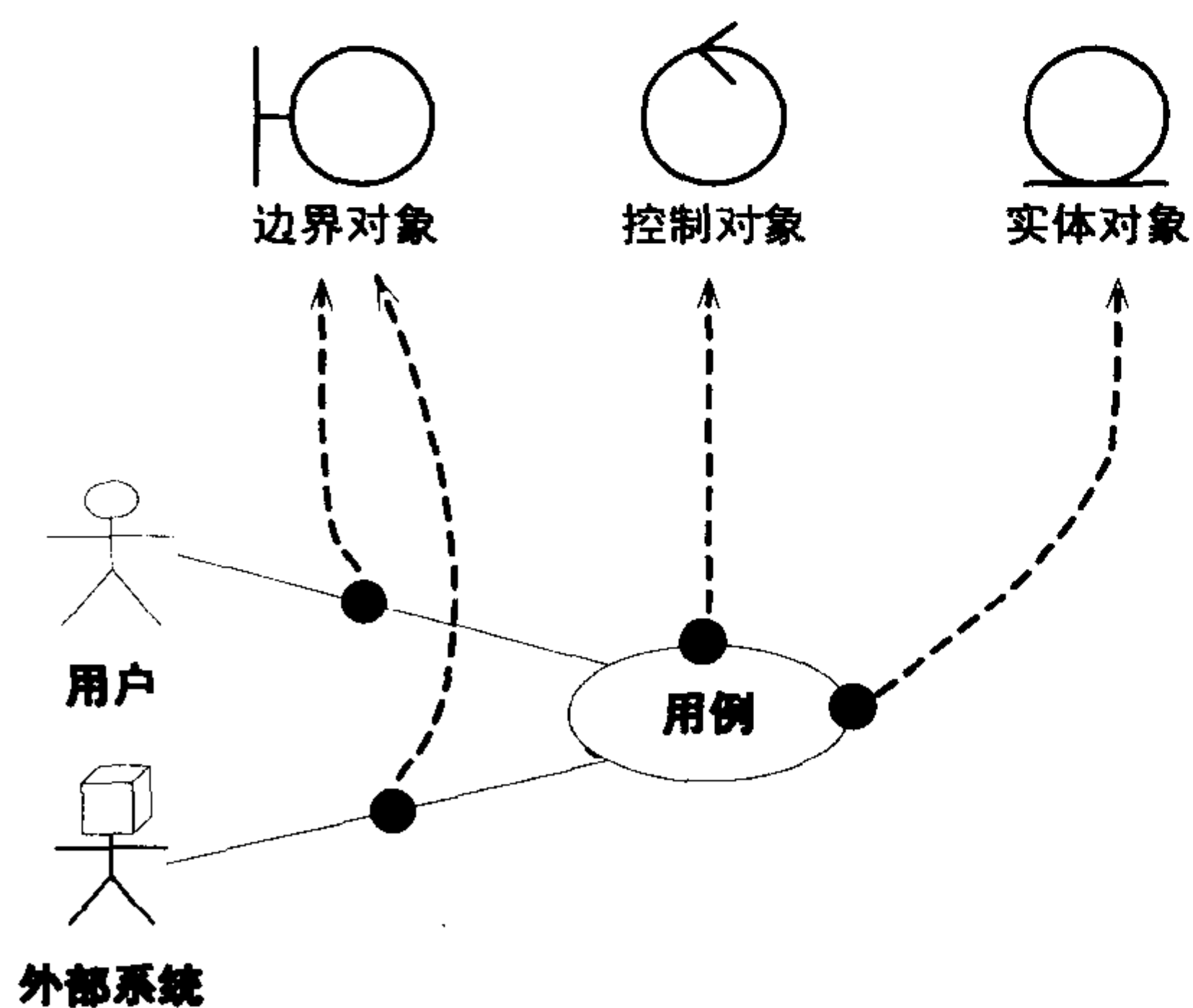


图 14-7 从用例到鲁棒图

将对不同用例进行分析得到的鲁棒图进行综合，可以得到和具体实现无关的理想化职责模



型。之所以说它是“具体实现无关的”，是因为这个模型关注职责的划分，而无论是交互、控制，还是信息维度的职责，都是和具体实现技术这一维是独立的（或称正交的），如图 14-8 所示。

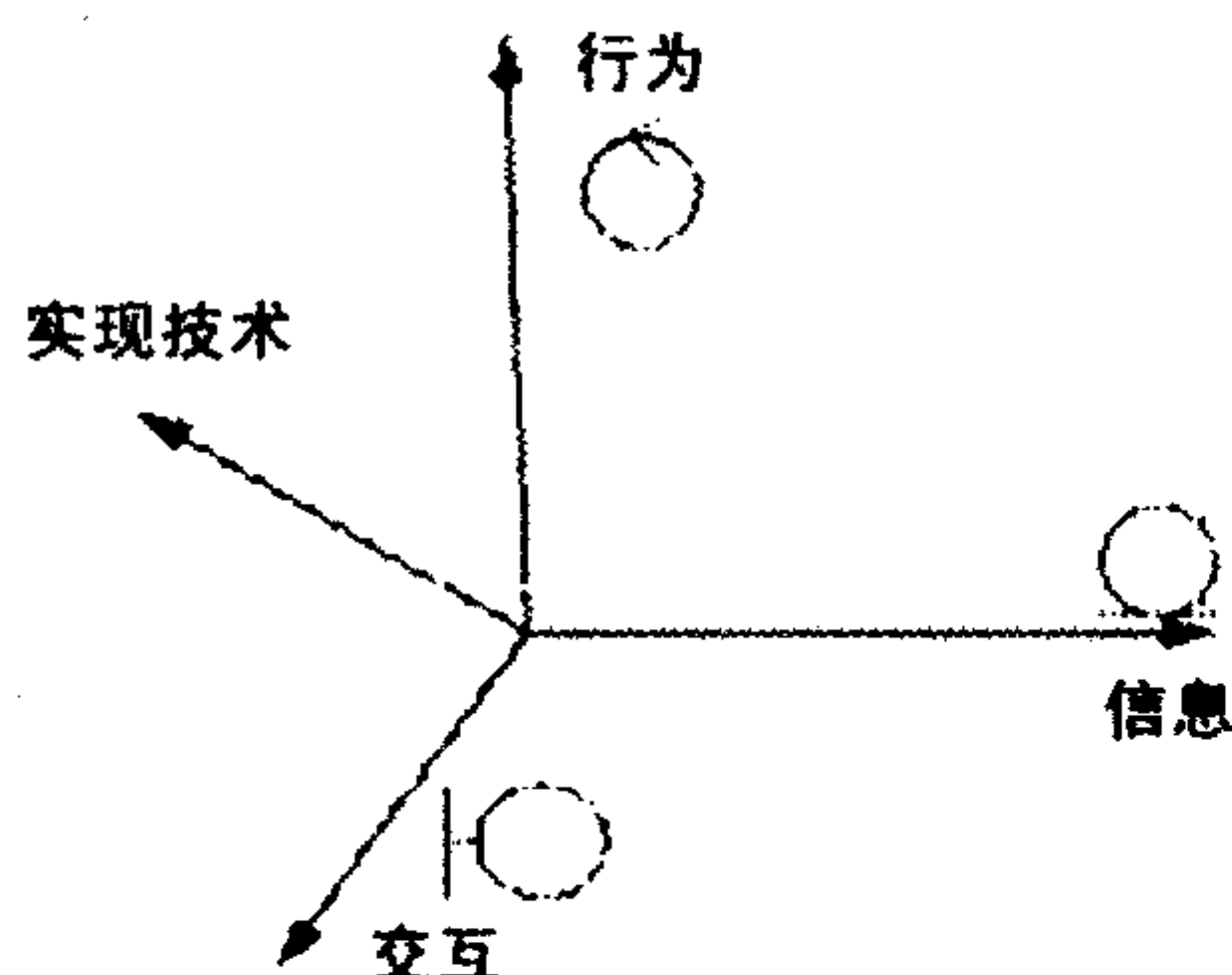


图 14-8 职责模型和具体实现技术无关

职责模型的技术无关性是个很大的优点，也正是“抛开”了大量具体技术细节，才使软件架构师更能够抓住概念性设计的本质，规划重大问题的关键解决策略。以高性能设计为例，无论是“对消息采用多线程并发处理”还是“将图片服务器独立出来以便进行专门的缓存和索引等加速处理”都需要对软件系统职责划分模型进行细化或调整。虽然为了满足“高性能”需求往往需要使用特定平台的非标准能力——这当然是和具体技术相关的，但本书认为，精心规划职责模型是获得高性能的根本，这也是为什么“将性能放在首位的软件系统”有时其架构与众不同的原因。

## 14.3 运用架构模式

所谓模式，就是解决相似问题的通用方法。难怪数学家说，世界是无限的，但模式是有限的。

### 14.3.1 架构模式简介

所谓软件架构就是关于如何构建软件的一些最重要的设计决策，这些决策往往是围绕将系统分为哪些部分、各部分之间如何交互展开的。

而软件架构模式呢？它是高度抽象的、适用于许多类似系统的、预先定义好的一种特殊的软件架构。

《面向模式的软件体系结构》则这样定义架构模式：

架构模式描述了软件系统基本的结构化组织方案。具体而言，架构模式提供了一套预定义的子系统，并规定了子系统的职责，以及子系统间关系的组织原则和组织指南（Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.）。

### 14.3.2 架构模式的经典分类

本书将表 14-1 所示的架构模式分类方法称为“经典分类方法”，这意味着它有着广泛的影响，但未必是当前实践中所采用的方式了。

表 14-1 架构模式的经典分类

类别	名称	例子
调用—返回	分层 (Hierarchical layers)	OSI 7 层网络协议
	主程序—子程序 (main program and subroutine)	太多了
	面向对象 (OO systems)	太多了
独立组件	通讯进程 (communicating processes)	某些自控系统
	事件系统 (event systems)	OS 微内核结构，MFC message 机制
虚拟机	解释器 (interpreters)	C++ Compiler，任天堂模拟器 on PC，CPU 软件模拟器 for 嵌入式开发
	规则为中心 (rule-based systems)	某些人工智能系统
数据流	批处理 (batch sequential)	某些老式 OS，金融系统日终处理
	管道—过滤器 (pipes and filters)	MS DirectShow，Unix CC（compiler/linker 等串起来一步步加工）
数据为中心	数据库 (database)	DB2，Oracle
	黑板 (blackboards)	某些人工智能系统



介绍这种方法的经典图书是《软件体系结构——一门初露端倪学科的展望》，读者可参考。

14.3.3 架构模式的现代分类

上面的架构模式的经典分类法，将面向对象作为一种架构模式，真是令人匪夷所思——因为面向对象开发方法应该和任何一种架构模式都没有直接关系才对。另外，MVC 架构模式如此流行，而上述分类方法也未提及……所以我们下面介绍另一种架构模式的分类方法，我们暂且称之为架构模式的现代分类方法（如表 14-2 所示）。

表 14-2 架构模式的现代分类

类别	名称	特点	例子
从混沌到结构	分层 (Layer)	<ul style="list-style-type: none"><li>· 直观的分而治之方式</li><li>· 层的重用</li><li>· 依赖性局部化</li><li>· 可替换性</li></ul>	通信协议栈 Java 虚拟机 很多应用系统
	管道—过滤器 (Pipes and Filters)	<ul style="list-style-type: none"><li>· 重用性好</li><li>· 便于重组新策略</li><li>· 交互性差</li><li>· 适用于复杂处理</li></ul>	编译器 Unix Shell DirectShow 通信协议解析
	黑板 (Blackboard)	<ul style="list-style-type: none"><li>· 算法与数据分离</li><li>· 耦合度大</li><li>· 适用于某些 AI 系统</li></ul>	人工智能系统
分布式系统	代理者 (Broker)	<ul style="list-style-type: none"><li>· 屏蔽复杂性</li><li>· 支持互操作</li><li>· 适用于分布式系统</li></ul>	CORBA RMI 交叉编译系统
交互式系统	MVC (Model-View-Controller)	<ul style="list-style-type: none"><li>· 灵活性</li><li>· 广泛流行</li></ul>	很多
	PAC (Presentatin-Abstraction-Control)	<ul style="list-style-type: none"><li>· 灵活性</li><li>· 复杂</li></ul>	人工智能系统
适应性系统	微内核 (Microkernel)	<ul style="list-style-type: none"><li>· 适应变化</li><li>· 支持长生命周期</li><li>· 复杂</li></ul>	OS JBoss Eclipse
	基于元模型 (Meta-level Architecture)	<ul style="list-style-type: none"><li>· 适应变化</li><li>· 支持长生命周期</li><li>· 不易实现</li></ul>	反射框架 Spring MDA

区别很明显，如果说架构模式的经典分类方法是根据交互机制进行的话（调用、发消息或是

共享数据), 那么现代方法就是将架构模式的目标作为分类的依据。显然, 后一种分类方法更利于软件架构师在实践中挑选合适的架构模式。

最后, 介绍这种方法的经典图书是《面向模式的软件体系结构》, 读者可参考。(有趣的是, 这本书和《软件体系结构——一门初露端倪学科的展望》都是在 1996 年出版的, 所以“经典派”和“现代派”之别完全是本书根据它们贴近实践的程度“杜撰”的。)

### 14.3.4 分层

分层架构极为流行, 似乎毋庸多言。但实践中的分层架构早已不是书本上的分层架构了。

先说说经典的分层架构。分层架构的最大优点是将整体问题局部化, 把可能的变化分别封装在不同的层中。最终, 系统被规划为一个单向依赖的分层体系, 利于修改、扩展和替换。具体而言, 各层之间的单向依赖又可以分为两种: 严格的分层架构要求第  $n$  层只能被第  $n+1$  层调用, 与此相对的是不严格的分层架构, 第  $n$  层可以被位于其上层的任意一层调用。如图 14-9 所示。

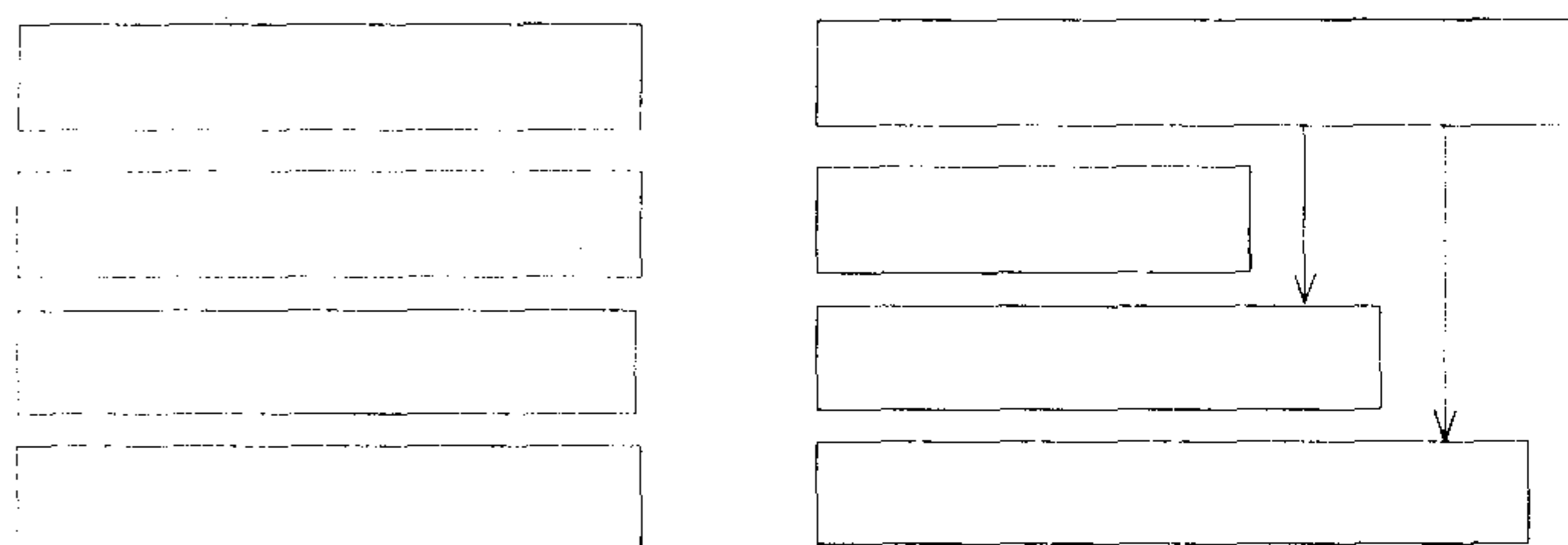


图 14-9 严格分层与一般分层

再说说我们大多数人是如何运用分层架构的。简言之, 分层架构在实践中“退化”了。如果说任何一种架构模式都关注职责划分和交互机制两方面的话, 那么我们常用的“分层架构”往往只关注职责划分。这种做法本身并无任何害处, 利用分层来划分职责也很有效, 但应注意架构设计工作不要就此罢手, 形成“名不副实的分层架构”设计方案(参见第 8 章“软件架构要设计到什么程度”)。

### 14.3.5 MVC

MVC 架构是随着 Smalltalk 语言的发展提出的, 而交互式应用系统的发展促进了 MVC 的流行。采用 MVC 架构的软件会包含这样三种组件: Model、View、Controller。同时, 为了达到一定的目的, 这三种组件必须通过交互来协作, 比如: View 创建 Controller 之后, Controller 根据用户交互调用 Model 的相应服务, 而 Model 会将自身的改变通知 View, View 会读取 Model 的信息以更新自身。如图 14-10 所示。



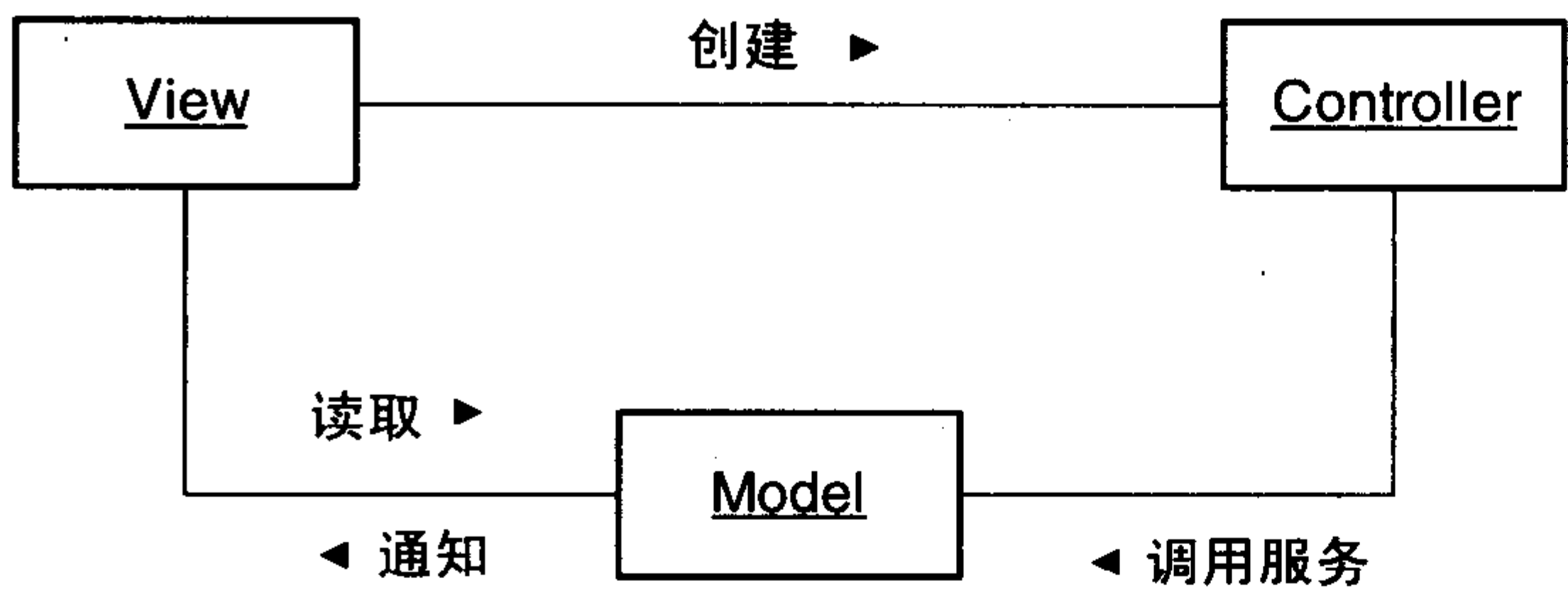


图 14-10 MVC 架构模式

14.3.6 微内核

微内核架构“生来”就是为了适应变化的。为此，微内核架构不仅将应用相关部分与通用部分分离，而且又将通用部分进一步分离成一个最小化的基本服务内核和众多扩展服务。微内核仅提供一个基本服务的最小集合，这些基本服务屏蔽了其下层复杂环境的影响、并以规范的接口提供给上层的“增值服务”使用，于是所有这些附加的增值服务都和复杂环境无关。

微内核所提供的基本服务必须是完备的，从而实现一个虚拟机。这个虚拟机不仅屏蔽了下层的复杂性，而且建立了更抽象、更易用的一个“新机器”供其上层使用。

我们在讨论微内核架构的优点之前首先说明其缺点。这是因为，相对与同种情况下采用其他架构模式而言，采用微内核架构有明显的缺点，每一个软件架构师决定采用微内核架构之前都必须慎重考虑这些，确保在获得微内核架构优势的同时，能够接受或避免它的常见缺点：

- 微内核架构最大的缺点是设计和实现的复杂性，这是因为微内核提供的能力必须少而精，但又必须完备，否则就不能提供想要的完整“机制”；
- 另外，微内核架构往往比同类情况下的其他设计性能低，要解决此问题必须精心设计与微内核架构所依据的职责模型相匹配的性能优化策略。

微内核架构的优点是诱人的，这也正是这种在多年前仅被用于操作系统等系统软件的架构模式被越来越多的应用级系统所采用的原因：

- 可扩展性。微内核本身是稳定的，所有附加功能均可以通过开发新的附加服务的形式进行扩充。而且，微内核提供了封装底层功能的更易用的虚拟机，附加服务的开发可以在较高的层次上开始；
- 可移植性。附加功能只依赖于微内核，它们没有移植性的开销。只需修改微内核的移植相关部分即可；
- 延长软件系统的生命周期。微内核架构可以接纳一些当前不可预见的新技术，以及商业层面的新功能的挑战。

微内核架构适用于开发难度大、预期生存周期长、成本高的软件系统，通过采用微内核架构

可以很好地解决这三个问题：

- 微内核将底层差异进行封装，构建了一个抽象度更高的虚拟机，降低了其他服务开发的难度；
- 生存周期越长，需要应对的变化就越多。微内核架构屏蔽了下层变化，为上层服务提供了更稳定的抽象服务，利于大规模扩展的需要；
- 正如我们看到的，通过开放性可以缓解成本高的问题。将微内核实现的基础服务开放，可以吸引大批开发者来增强你的产品。

### 14.3.7 基于元模型的架构

随着业务和技术变化的加剧，对随需应变的要求越来越多了，基于元模型的架构和微内核架构一样，都是以适应变化、支持长生命周期为主要目标的架构模式。

基于元模型的架构包含两个层：基本层和元层次。如图 14-11 所示。

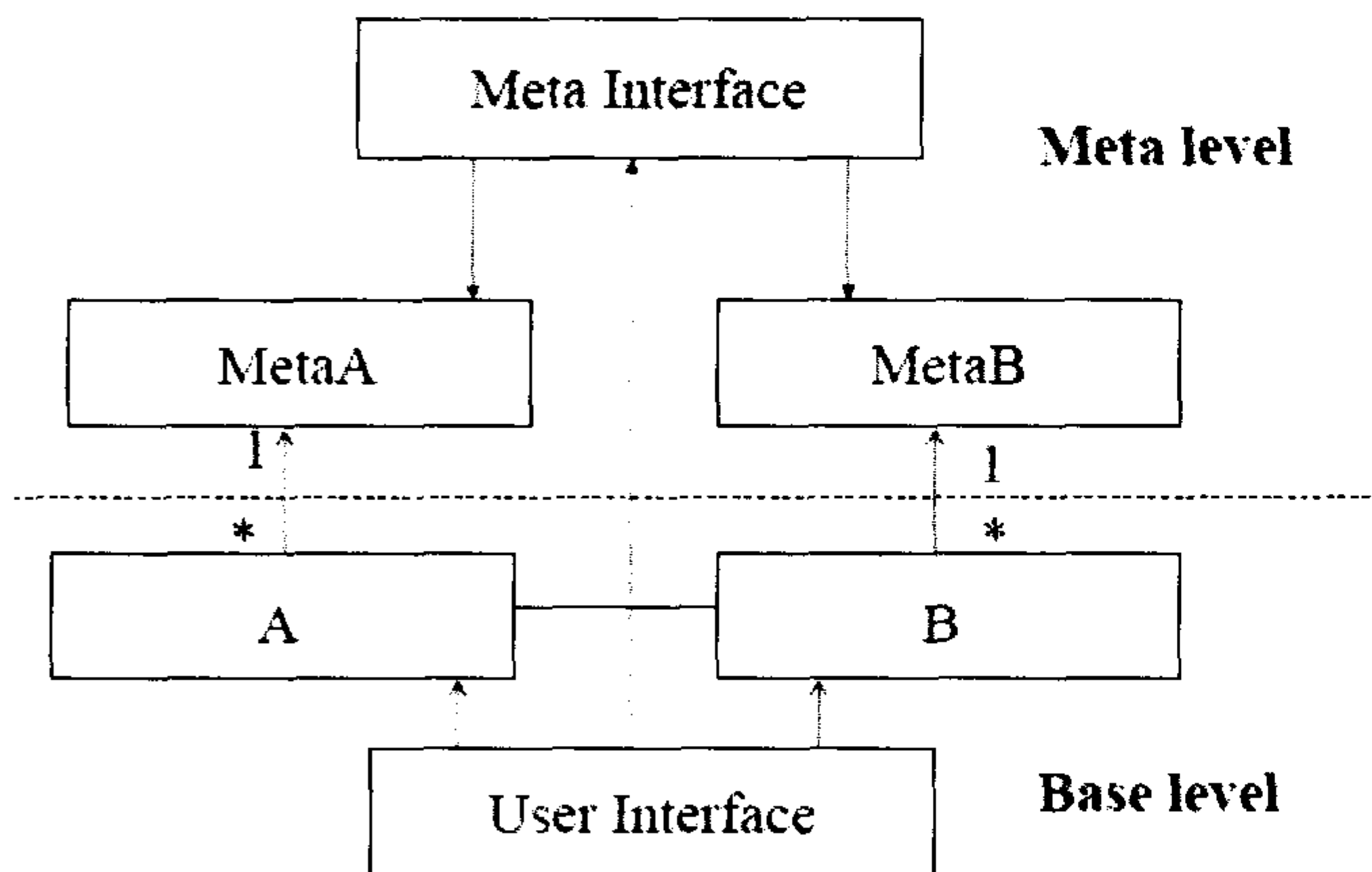


图 14-11 基于元模型的架构（图片来源：Jon Kuhl 的讲义）

基本层定义应用，它类似于我们要编写的程序，只不过“程序”的某些部分（或全部）本来可以直接调用某个对象 A 的，却改为通过这个对象的元对象 MetaA 进行调用，从而解除了应用对对象 A 的依赖。

元层次包含一组“元对象”，这些元对象组成的模型叫做“元模型”，元模型是对基本层中的“一切”的抽象。元模型提供了软件本身的结构和行为的“自描述”。一般而言，元模型中的元对象应分别从三个方面刻画基本层：结构、行为、状态。不难理解，元层次的实现少不了类型感知、运行时方法调用等技术的支持。

### 14.3.8 管道—过滤器

采用管道—过滤器架构模式可以支持很复杂的数据处理问题，在编译器、图像处理、通信协议解析方面有广泛应用。图 14-12 展示了编译器是如何运用这一架构模式的。

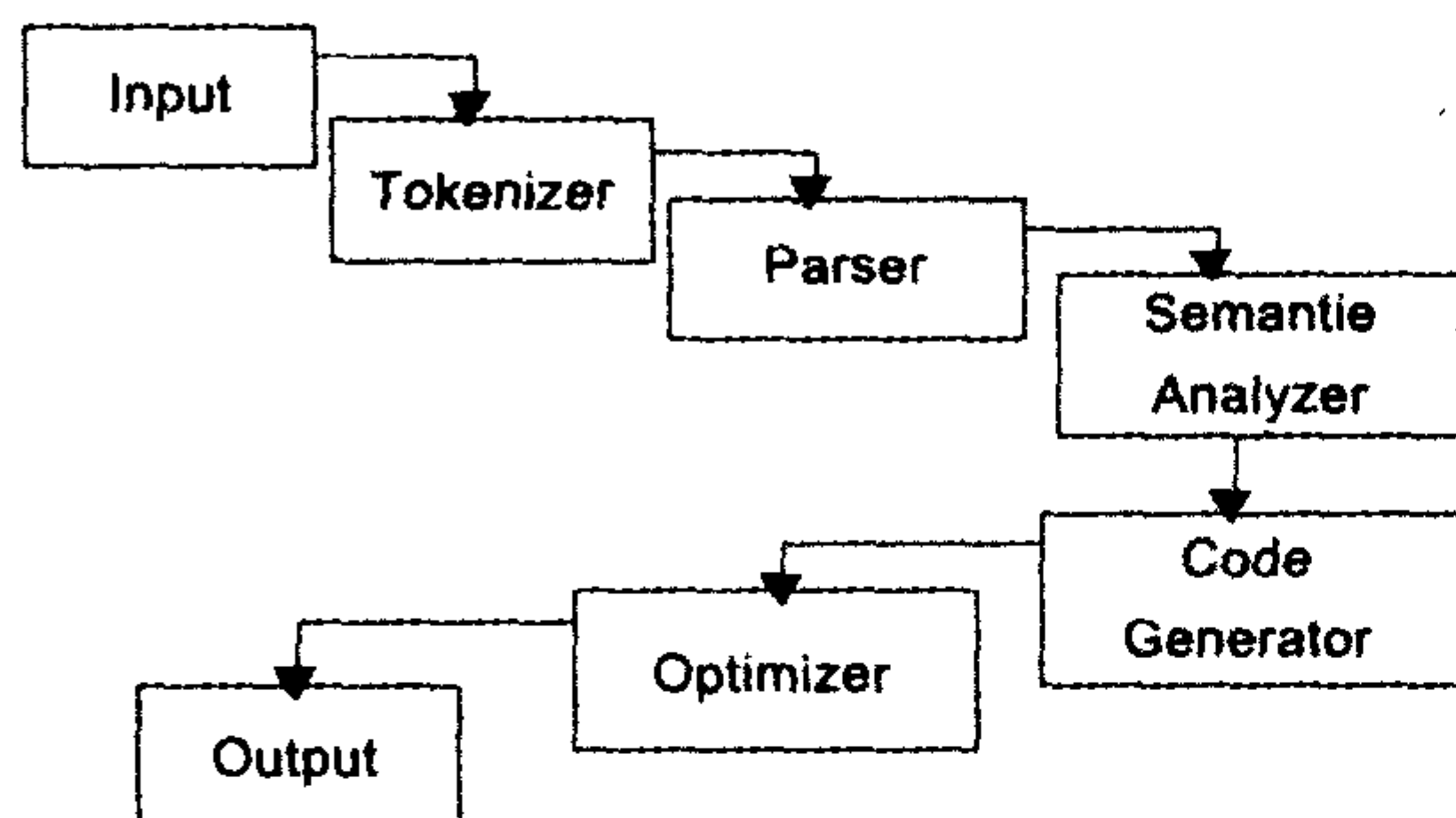


图 14-12 编译器的管道—过滤器架构

管道—过滤器架构的缺点是交互性差，就像批处理的交互性差一样。按照“边界对象、控制对象、实体对象”的观点来看，这个缺点正是因为没有“边界对象”的缘故——它是控制对象和实体对象的间插排列形成的一条流水线。

## 14.4 PM Tool 实战：概念性架构设计

上一章的“PM Tool 实战”中，我们确定了对架构设计关键的需求子集。下面我们根据这些关键需求进行概念性架构的设计。

### 14.4.1 进行鲁棒性分析

首先，对“添加项目任务”用例进行鲁棒性分析。图 14-13 展示的鲁棒图代表了对“添加项目任务”功能的初步设计。例如，在持久化数据方面，我们识别出了资源、项目、任务、任务分配等实体。

接着，我们对“从 HR 系统导入资源”用例进行鲁棒性分析。如前所述，这个用例之所以对架构设计比较重要，是因为它涉及了和外部系统是如何交互的，而其他用例没有涉及此方面。如图 14-14 所示，初步设计要求将 HR 系统的远程访问配置保持在相关“配置信息”中，它可以是数据库也可以是普通文件；另外，“资源列表”是在两个系统之间传递的数据，数据的格式应预先定义好供发送和解析使用。



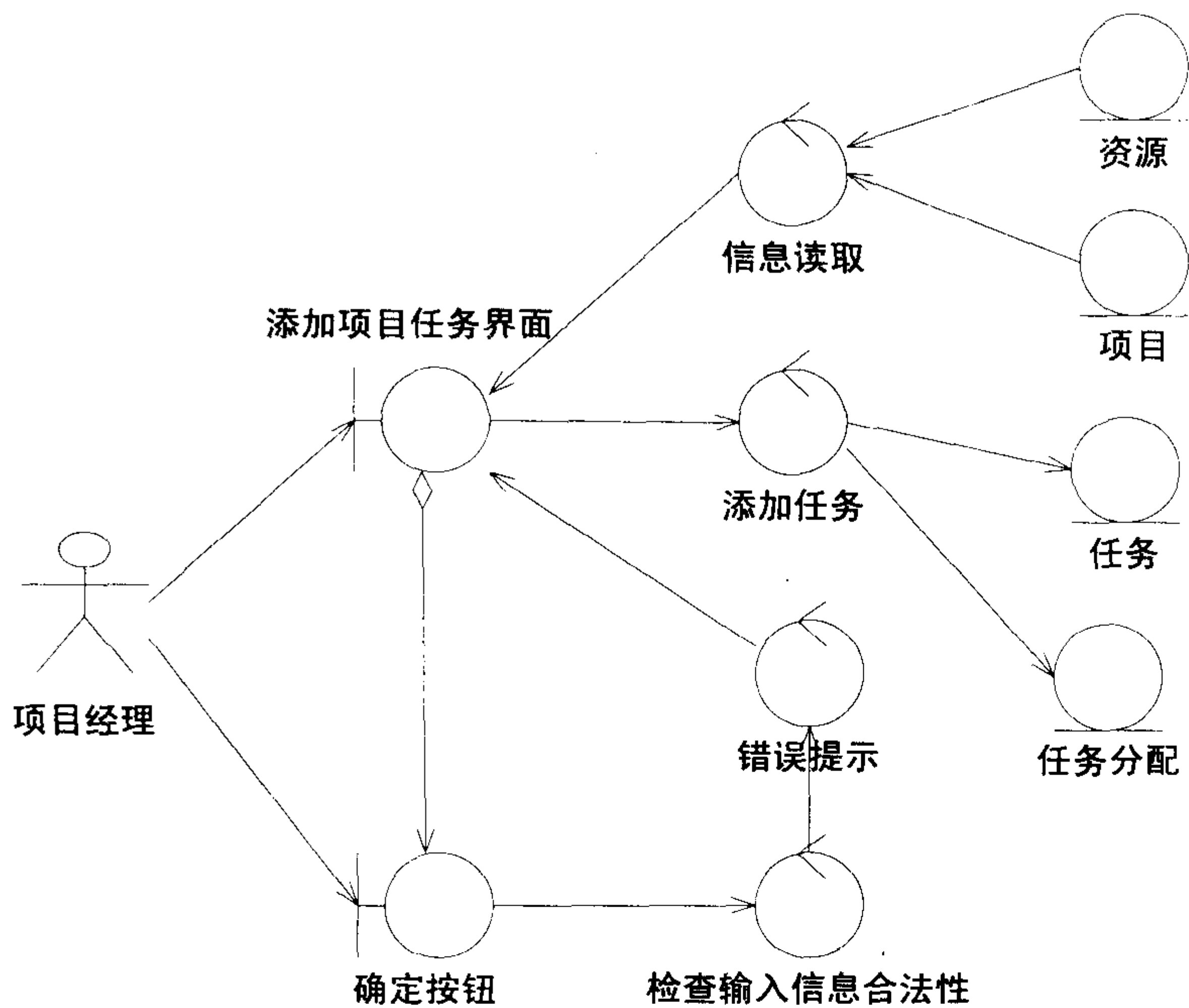


图 14-13 实现“添加项目任务”的初步设计

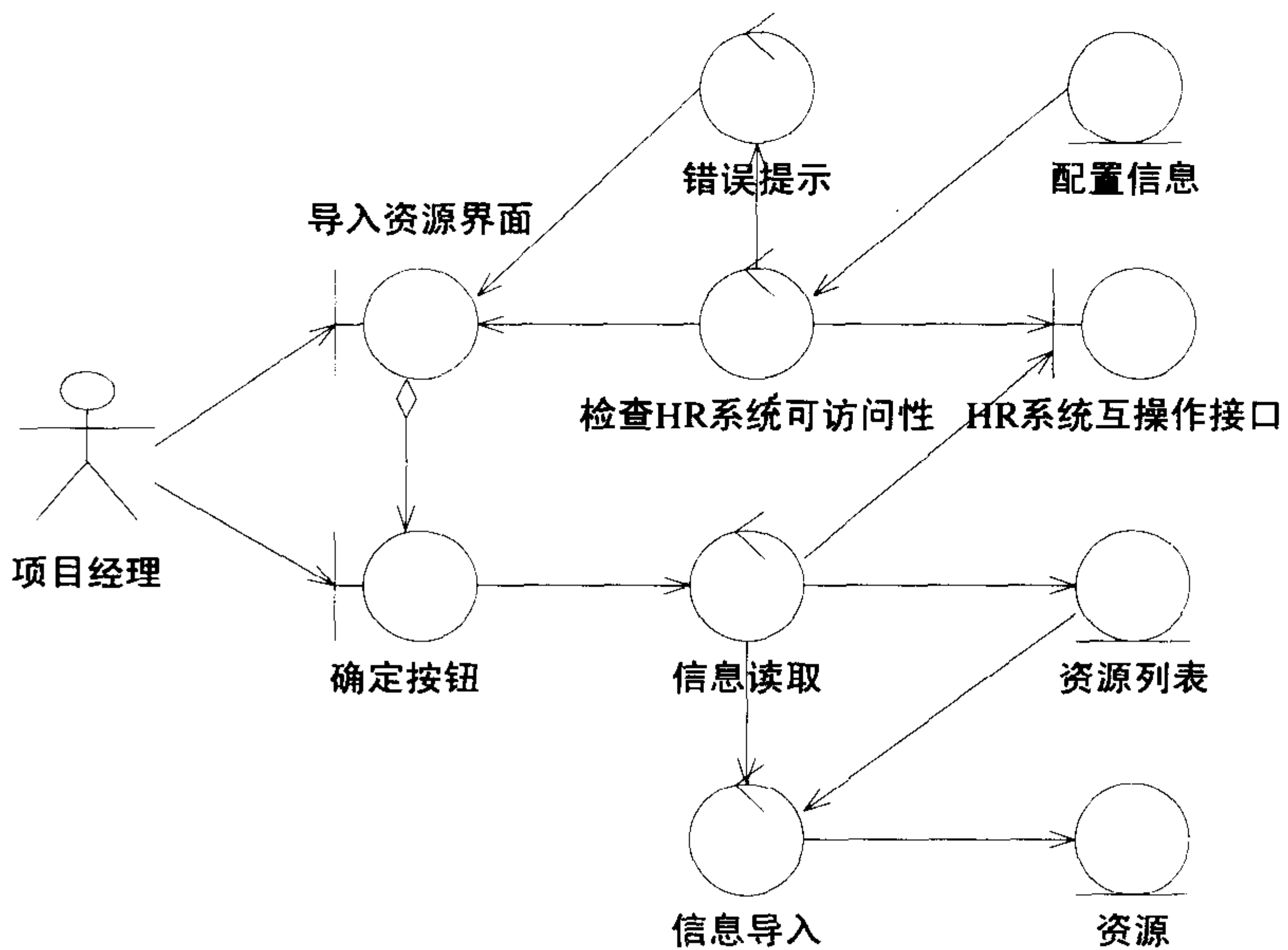


图 14-14 实现“从 HR 系统导入资源”的初步设计

一般而言，应对所有对架构起关键作用的用例进行鲁棒性分析，在此不再一一列举。

14.4.2 引入架构模式

将鲁棒性分析的成果综合起来，我们的脑中就有了一个理想化的解决方案，其中识别出了有足够代表性的职责。下面，我们就可以考虑引入何种架构模式，来组织零散的职责元素，并明确关键的交互机制了。图 14-15 展示了在初步设计的基础上，是如何引入分层架构模式的。例如，“集成层”应负责和外部系统的交互，并将交换数据的格式等交互细节屏蔽。

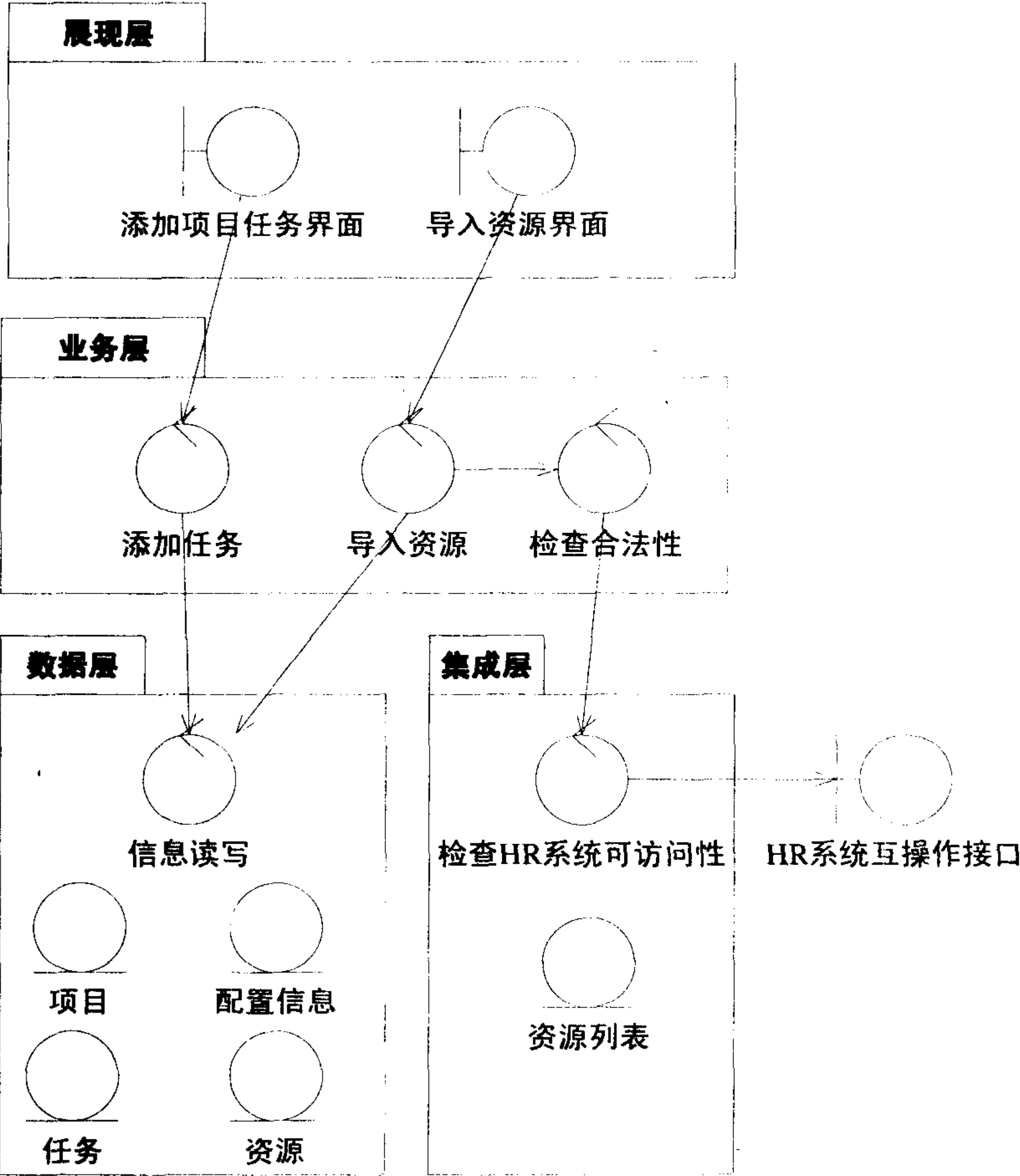


图 14-15 引入分层架构模式

14.4.3 质量属性分析

为了确保非功能需求被满足，在概念性架构设计中必须将希望达到的质量属性细化为具体的场景，并由此制定高层的架构设计决策。表 14-3 展示了我们的思考设计过程，它运用了“属性—场景—决策”表方法，该方法在下一章将进行专门介绍。

表 14-3 “属性—场景—决策”表

属性	场景	决策
商业需求	项目成员可能到客户现场进行开发	PM Tool 应采用 B/S 结构
	PM Tool 的客户群工作的平台多样化	PM Tool 应采用 B/S 结构
互操作性	从 HR 系统中导入员工信息	PM Tool 应： 公开期望的 Web 服务接口 公开资源列表文件格式要求 提供 HR 服务地址的配置界面
	集成对 CVS 和 Subversion 的访问	定义和具体配置管理服务器无关的接口， 采用 Adapter 模式分别实现不同服务器的访问
跨平台运行	服务器端可能运行在 Windows、Unix 或 Linux	采用 Java 开发便于获得跨平台特性
	数据库服务器应支持 MySQL 和 Oracle	采用 OR Mapping 框架

14.4.4 设计结果

最终，PM Tool 的概念性架构设计完成（如图 14-16 所示）。我们采用分层架构将不同职责分离，例如集成层的职责包括访问外部系统、为外部访问提供接口和负责交换数据的格式等。

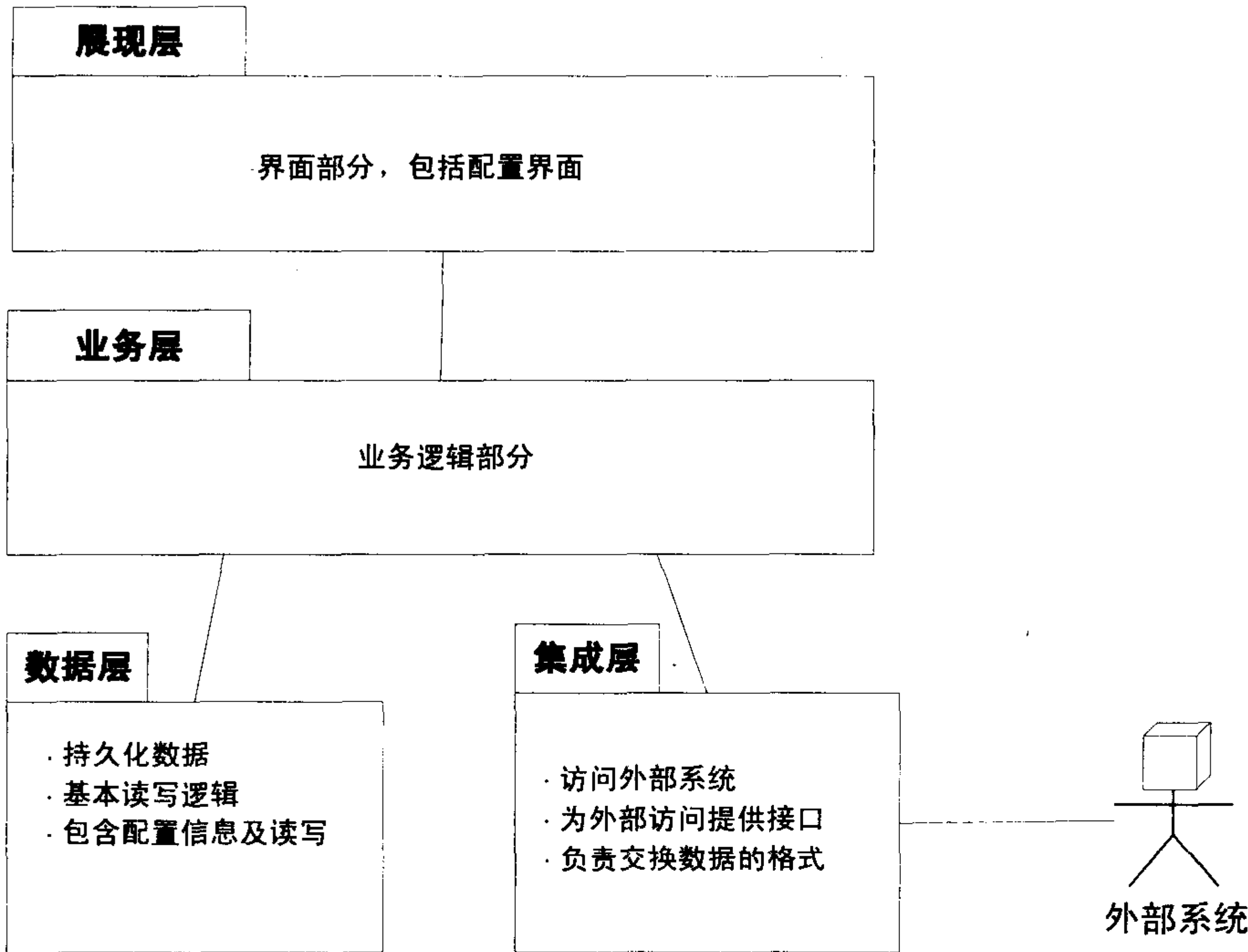


图 14-16 PM Tool 的概念性架构

### 14.5 总结与强调

概念性架构应为系统的关键设计目标负责，对规模很大的软件系统非常有意义。另外它使软件架构师不必一开始就关注太多具体技术，符合人们解决问题的规律。



## 第 15 章 质量属性分析

为了提高综合客户满意度以及对不同质量属性的满意度，必须考虑在计划 and 设计产品时的不同质量属性。

——Stephen H. Kan, 《软件质量工程》

质量属性很难定义，但它们经常可以区分产品是只完成了其应该完成的任务呢，还是使客户感到很满意……优秀的软件产品反映了这些竞争性质量属性的优化平衡。

——Karl E. Wiegers, 《软件需求（第 2 版）》

很少有需求文档能够以一种可测试的细节捕获系统所有的质量需求。现实情况是设计师通常不得不填补空白并协调冲突。

——Len Bass, 《软件构架实践（第 2 版）》

在我们当中，有不少人一厢情愿地认为：只要所开发出的系统完成了用户期待的功能，项目就算成功了。但这并不符合实际。

忽视包括质量属性需求在内的非功能需求是很要命的。试想，新浪发布的一个重大新闻会引起怎样的突发性大规模访问，如何保证用户响应时间接近常量（例如在性能达到“拐点”时启用专门的缓存机制，如图 15-1 所示）？试想，一个网上售书平台如果想提供“在所有图书的目录中进行搜索”的功能，应如何保证性能？例子还有很多……

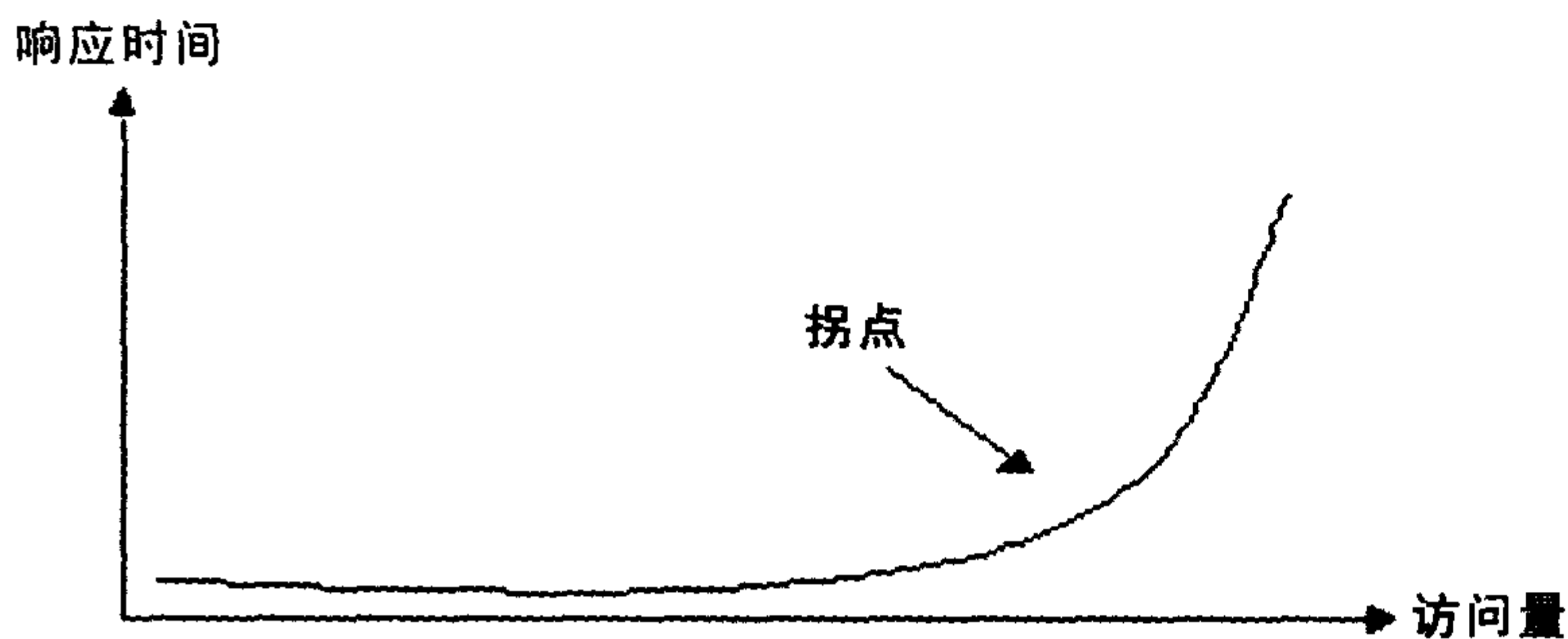


图 15-1 可伸缩性

我们周围的现实也的确如此。为什么不少软件产品推出不久就要重新设计（美其名曰“架构重构”）？往往不是因为系统“不能用”，而是由于系统架构“太拙劣”的原因——从难以维护、运行速度太慢、稳定性差甚至当机频繁，到无法进行功能扩展、易遭受安全攻击等，不一而足。由此看来，软件的质量属性需求是不容忽视的，否则在大量的成本投入之后，很可能落得“失败”或“赔钱”的结果。

然而，软件的质量属性需求很“飘”，常常令架构师难以把握。如果缺乏足够的方法指导，即使勉强制定了设计决策也会觉得缺乏信心。

本章要讨论的主题正在于此：如何进行质量属性分析，如何明确必须满足的质量属性场景，如何制定出相应的架构设计决策。

## 15.1 质量属性需求基础

软件架构师面对纷繁复杂的需求，必须对需求分类、需求折衷和需求变更的知识和规律有透彻的了解，从而把握大局、抓住重点，做出最合适的架构设计决策。至于质量属性需求，架构师必须了解以下几方面的知识：

- 软件质量属性可以划分为运行期质量属性和开发期质量属性两大类。这一点很重要，因为为了满足性能、持续可用性等运行期质量方面的要求，架构师必须深入研究软件系统运行期间的情况，合理划分系统不同部分的职责，权衡轻重缓急，并制定相应的并行、分时、排队、缓存、批处理等设计决策；而要满足可扩展性、可重用性等开发期质量方面的要求，则要求架构师深入研究软件系统开发期间的职责划分、变化隔离、框架使用、代码组织等情况，制定相应的设计决策。详细内容请参考第 10.3.1 节“软件需求的类型”；
- 各类需求对架构设计的影响不同，不同质量属性对架构也要求各异。例如，为了获得高可移植性，架构设计中必须考虑对硬件和平台相关特性进行封装和隔离。再例如，精心规划职责模型是获得高性能的根本，这就是为什么“将性能放在首位的软件系统”有时其架构与众不同的原因。详细内容请参考第 10.3.2 节“各类需求对架构设计的影响”；
- 众多质量属性需求之间往往会有冲突，我们必须权衡。例如，可移植性促进可测试性，但可测试性却不能促进可移植性（也不会降低可移植性），可移植性可能对可维护性造成负面影响……详细内容请参考第 10.3.5 节“质量属性需求与需求折衷”。

## 15.2 质量属性分析的位置

质量属性分析是概念性架构设计的重要步骤（如图 15-2 所示）。概念性架构包括一些高层次的设计选择，对未来软件系统的质量和性能都起着关键作用。设计概念性架构的第一步是分析关键用例的用例规约，运用鲁棒图构造系统理想化的职责模型。接下来，明确架构模式，确定交互机制，形成初步的概念性架构。最后，还要通过质量属性分析，制定出满足非功能需求的高层设计决策，并根据这些设计决策对此前的工作成果进行增强、调整，以保证概念性架构体现这些设计决策。

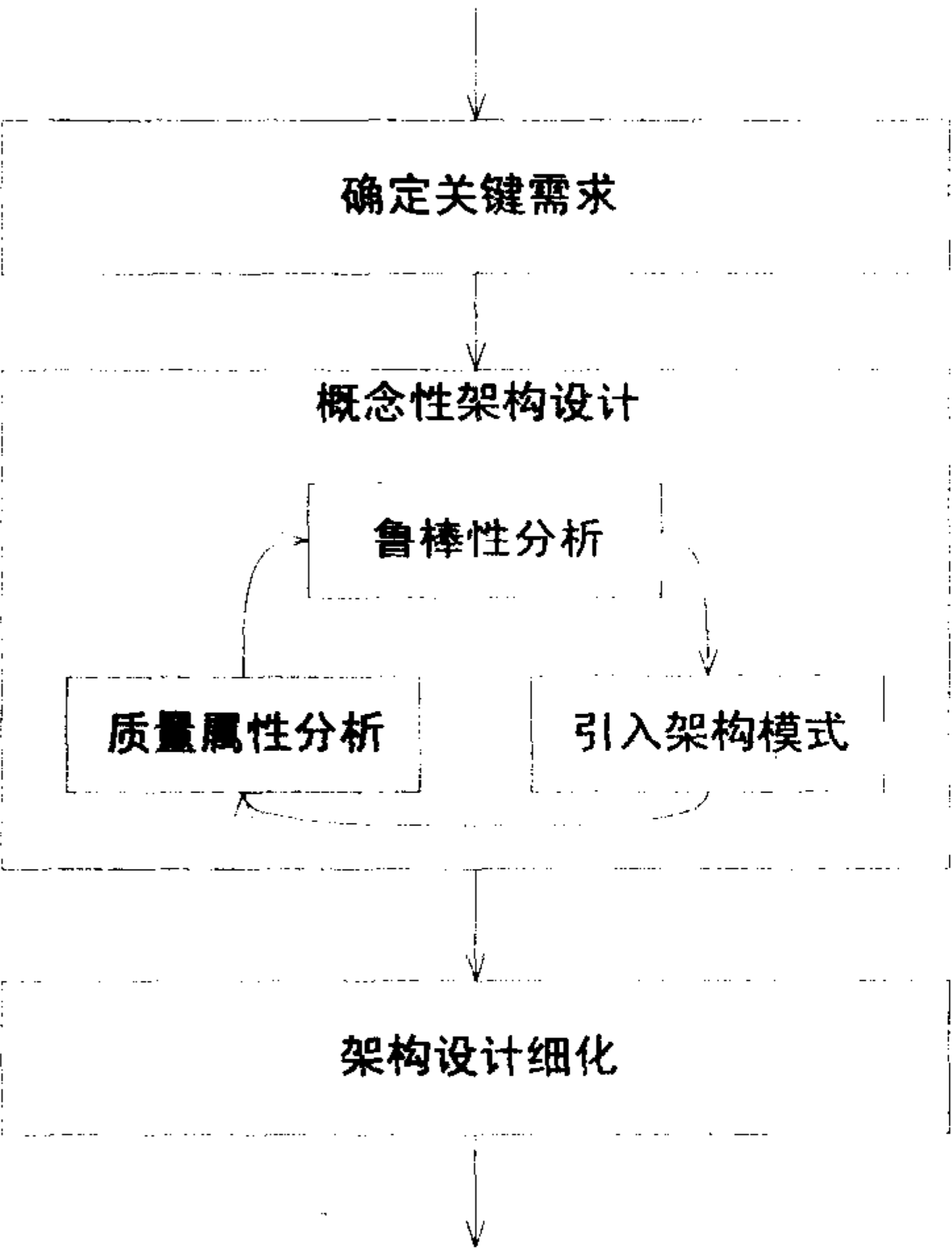


图 15-2 质量属性分析的“位置”

上游的“确定关键需求”活动会甄选出包含关键的功能需求、关键的质量属性要求，以及商业层面的目标和限制在内的一个需求子集。在概念性架构设计时，功能需求是鲁棒性分析最主要的输入，而质量属性需求是质量属性分析的最主要的输入。

## 15.3 利用“属性—场景—决策”表设计架构决策

### 15.3.1 概述

所谓质量属性分析，是软件架构师对软件系统要达到的质量属性需求进行分析、制定相应软

件架构设计决策的过程（如图 15-3 所示）。有经验的架构师知道把有限的时间用到什么地方，他们运用“关键需求决定架构”的策略，使质量属性分析的“输入”集中到关键质量属性需求。

“属性—场景—决策”表方法提倡通过一组具体场景将要达到的质量属性需求目标细化，再根据这些实实在在的场景制定架构决策。“属性—场景—决策”表方法的优点是可操作性强、符合人们思维的规律，同时还为评估软件架构设计质量提供了“可评测的”标准。笔者的一位同事戏称这种方法改变了他对架构设计的印象：“架构设计似乎不再是把大象放进冰箱了。”

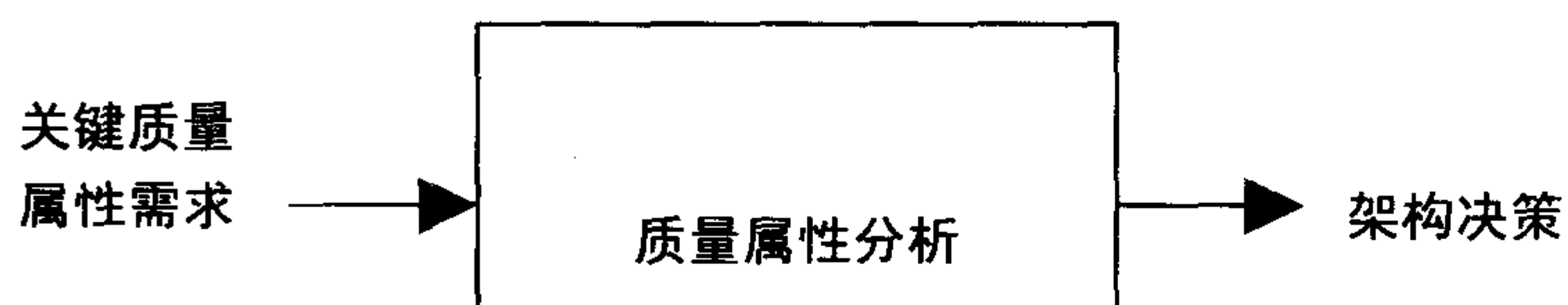


图 15-3 质量属性分析的“任务”

### 15.3.2 “属性—场景—决策”表方法

我们都知道，用例是一个连贯的功能单元，其具体形式是系统执行的一整套活动序列（既包括主线行为也包括对异常情况的处理），它能产生对特定参与者有价值的、可观察到的结果。而场景是上述“一整套活动序列”的特定执行路径。场景之于用例，正如对象之于类，前者是后者的实例。

随着实践的深入，我们发现有必要将“场景”的定义进一步放宽——至少用于“质量属性分析”的场景技术是这样。用于质量属性分析的场景技术并不拘泥于“用户使用”，它可以描述任何可能发出的情况，而不一定能产生对特定参与者可观察到的结果。

例如，对 PM Tool 而言，思考如何支持下列“场景”将有利于设计出灵活性高的架构：项目进度紧张时常把每周工作 5 天调整为每周工作 6 天，我应如何设计才能使项目成本核算等功能自动适应，而且这个“第 6 天”有时双薪有时三薪……

再例如，对银行的储蓄系统而言，一个有益的“易用性场景”是：一个中等业务水平的储蓄所工作人员能够在 5 个工作日内熟练掌握本系统 80% 的功能。

总之，质量属性场景既可以是用例场景（如银行卡用户在 ATM 上连续输入三次密码均错误），也可以不是（如 PM Tool 的用户调整项目日历）。图 15-4 展示了这一点。



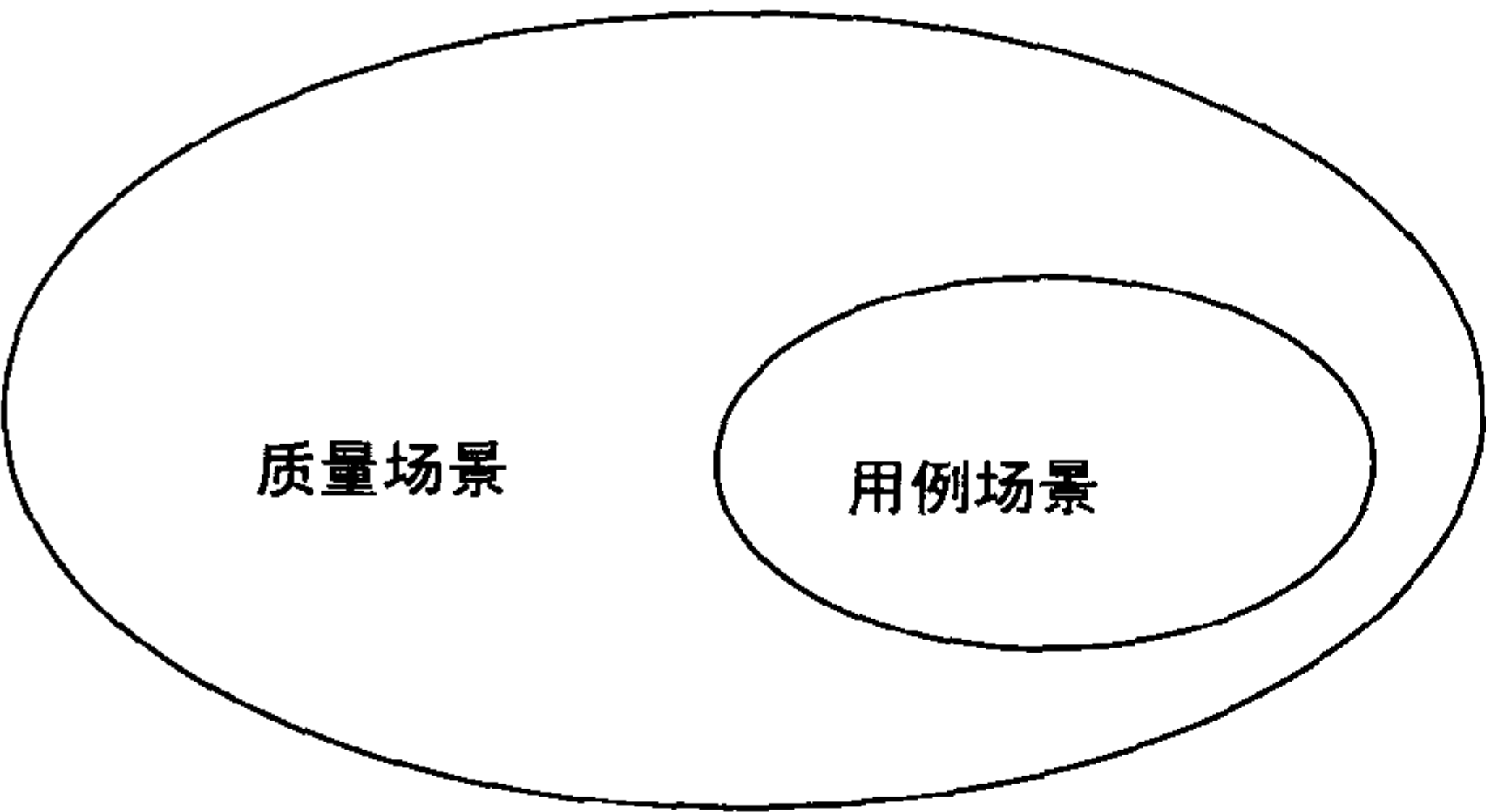


图 15-4 质量场景与用例场景的关系

一个笼统界定的“质量属性需求”往往让软件架构师无从设计，而将质量属性要求细化为一组具体的质量属性场景将有利于有针对性地制定软件架构设计决策。图 15-5 所示的“属性—场景—决策”表就是一种不错的思维工具。

属性	场景	决策
质量属性	质量场景	架构决策
	质量场景	架构决策
	质量场景	架构决策
质量属性	质量场景	架构决策
	质量场景	架构决策
	质量场景	架构决策

图 15-5 “属性—场景—决策”表

可以这么说，“属性—场景—决策”表方法使软件架构设计的决策过程从“黑盒”变成了“灰盒”，这样做有以下好处：

- 可操作性强。质量属性需求是笼统的目标，而一组质量场景使之明确化了，利于软件架构师有针对性地进行设计；
- 避免过度设计。单凭经验为高质量属性而设计很容易造成过度设计，即引入的很多抽象和机制是不必要的，平白增加了设计的复杂性。借助“属性—场景—决策”表方法很容易对质量场景进行评估，通过权衡场景发生的概率和遗漏它的代价，可以决定是否应满足该场景的要求；

- 便于系统升级时参考。当系统架构不能适应新要求时，往往要对架构进行重构，此时软件架构师常犯的毛病是过于强调系统架构的缺点，而将过去的架构设计全盘否定，这样可能造成设计出的新架构在解决了新问题的同时也失去了已有的优点。而如果将“属性—场景—决策”表文档化，则有利于避免上述问题。

当然，软件架构设计还没有变成“白盒”，因为架构设计依然既是科学又是艺术。所谓艺术，就是指没有固定不变的方法可以完全照搬，而是要依靠经验甚至灵感。例如，“属性—场景—决策”表方法中，质量场景的选择、架构决策的制定，都离不开架构师丰富的经验。

### 15.3.3 题外话：《需求文档》如何定义质量属性需求

在正确的时间做正确的事情对项目成功很重要。

笔者曾在“系统分析之窗”网站上看到过一篇题为《软件测试的革命》的好文章，其中写道：

Marick 很清晰地表述了他的观点：“我并不想写出一套用于捕捉用户愿望的需求，取而代之的是，我要写出一套测试，一旦这些测试能够通过，产品就能使她满意。所以我放弃了需求编写的步骤，而直接把需求分析加入到测试的创建过程中去。”

这段话很有启发性，虽然用“写测试”代替“写需求”还远不是现实，但我们相信“可测试的需求”是发展方向。

我们反观“将质量属性需求细化为一组质量场景”这一方法，发现实施它的最佳时机是需求分析期间，而不是架构设计之时——即《需求文档》应将质量属性需求细化到一组具体场景；因为需求分析期间和客户接触频繁，是发现质量场景的最好时机；而且在需求中将质量场景细化也为系统测试人员提供了测试标准。当然，由于质量场景不同于一般的只涉及行为需求的用例，而是有许多问题具有技术背景，所以，这一活动应该由软件架构师配合需求分析师一起完成。

## 15.4 PM Tool 实战：可扩展性设计

PM Tool 系统实战从第 10 章开始到第 13 章，经历了需求分析、领域建模、确定关键需求等步骤。首先回忆一下 PM Tool 系统的关键质量属性需求（如表 15-1 所示）：

- 易用性。因为复杂的项目管理工具反而会影响项目开发的高效进行，这和开发 PM Tool 的初衷相违背，所以一定要强调高易用性；
- 互操作性。有和其他系统互操作的现实需求和潜在需求，比如“PM Tool 从 HR 系统获取雇员信息”是当前就要求的，而随着公司业务的发展，将整个项目的一些子任务外包给合作伙伴也是必须支持的模式，这就导致了“PM Tool 和合作伙伴的 PM 系统

交互数据”需求的产生；

- 跨平台运行。为了使用户能够自由选择操作系统和数据库管理系统等，PM Tool 必须具有跨平台运行的能力；
- 可扩展性。潜在的功能增强要求非常明显。

表 15-1 PM Tool 的关键需求

约束	运行期质量属性	开发期质量属性	
客户群工作的平台多样化 成本效益考虑 应考虑外包趋势 和其他系统交换数据	跨平台运行 易用性 互操作性	可扩展性	创建项目 查看项目信息 添加项目任务 从 HR 系统导入资源 发布通知

接下来，我们运用“属性—场景—决策”表方法，细化 PM Tool 的可扩展性需求，并制定相应的架构决策，如表 15-2 所示。

表 15-2 PM Tool 的可扩展性设计过程

属性	场景	决策
可扩展性 (灵活性)	甘特图绘制包可替换	运用 Adapter 模式隔离变化
	如果 1.0 版反应良好，在 2.0 版将引入更多种类的图表	采用 MVC 模式将展现层分离
	项目管理模型可能增强或改变	将领域层分离，并通过 API 访问
	.....	.....

15.5 总结与强调

非功能需求对软件架构的影响比功能需求更大，这一点已经被业界越来越多的人所认同。本章介绍的思维工具——“属性—场景—决策”表——可以帮助软件架构师以一种更透明、更可操作的方式完成从质量属性需求到质量场景的细化，最终根据具体的场景有针对性地设计架构决策。





## 第 16 章 细化架构设计

---

多视图方法不仅仅是架构归档技术，更是指导我们进行架构设计的思维方法。

——温昱，《运用 RUP 4+1 视图方法进行软件架构设计》

很多因素，例如数据库管理系统、分布式、具体程序设计语言、现有产品、性能和存储描述等，都不在分析模型中说明，而要在设计模型中说明。

——Ivar Jacobson，《软件复用：结构、过程和组织》

使用多重视图允许独立地处理各“风险承担人”：最终用户、开发人员、系统工程师、项目经理等所关注的问题，并且能够独立地处理功能性和非功能性需求。

——Philippe Kruchten，《架构蓝图：软件架构“4+1”视图模型》

有两个很现实的问题。

由于角色和分工不同，整个软件团队以及客户等涉众各自需要掌握的技术或技能存在很大差异，为了完成各自的工作，他们需要了解整套软件架构决策的不同子集。例如，负责部署和运营维护的系统工程师最关心软件系统基于何种操作系统之上、依赖于哪些软件中间件、有没有群集或备份等部署要求、驻留在不同机器上的软件部分之间的通信协议是什么等决策；而开发人员则最关心软件架构方案中关于模块划分的决定、模块之间的接口如何定义、已经架构规定的开发技术和现成框架等问题；……毕竟，将不同涉众关心的技术堆砌到一起，每个涉众都有可能看不懂了。这是第一个问题。

另一方面，软件架构必须围绕“如何构建软件”制定多方面的设计决策，可能涉及的概念有很多：逻辑层（Layer）、物理层（Tier）、子系统、模块、接口、进程、线程、消息、协议，等等。如果不能把这些“搅在一起”的问题在一定程度上“进行拆解”的话，对架构设计的开展很不利。这是第二个问题。

本书第 5 章介绍的基于 5 视图的架构设计方法，是解决上述两个问题的卓有成效的办法。本

章将讲述如何在概念性架构的基础上，运用 5 视图方法进行架构设计的细化。实际经验表明，越是复杂的系统，越是需要从多个方面进行架构设计，这样才能把问题研究和表达清楚。而提供不同的软件架构视图也便于交流和传递设计思想。总之，软件架构是团队开发的基础，运用基于 5 视图的架构设计方法可以比较明确地规定后期分头开发所必须的公共性的设计约定，从而为分头开发提供足够的指导和限制。

## 16.1 架构细化在软件过程中所处的位置

### 16.1.1 我们走到哪了

在本书的第 14、15 章中，我们讨论了概念性架构设计，确定了最为关键的设计要素和交互机制。但概念性架构是不可直接实现的，开发人员拿到概念性架构设计方案依然无法开始具体的开发工作。

接下来要进行的工作（即本章要讲的内容）是在概念性架构的基础上，运用更多具体的设计技术，设计出能够为实际开发提供更多指导和限制的实际架构。例如，实际架构重视通过子系统和模块来分割整个系统，并为每个子系统定义明确的接口；而概念性架构中只涉及抽象的组件——它们没有接口、只有职责，这些组件一般是处理组件、数据组件或连接组件中的一种。

软件架构是接下来要进行的大规模开发的基础。软件架构中包含了软件系统如何组织等关键决策，模块的技术细节被局部化到了小组内部，不同小组的成员需要精通的技术各不相同，并且内部的细节不会成为小组间协作沟通的主要内容，这样就理顺了沟通的层次。

图 16-1 总结了概念性架构、实际架构、开发实现的关系：上下承接，不断细化。

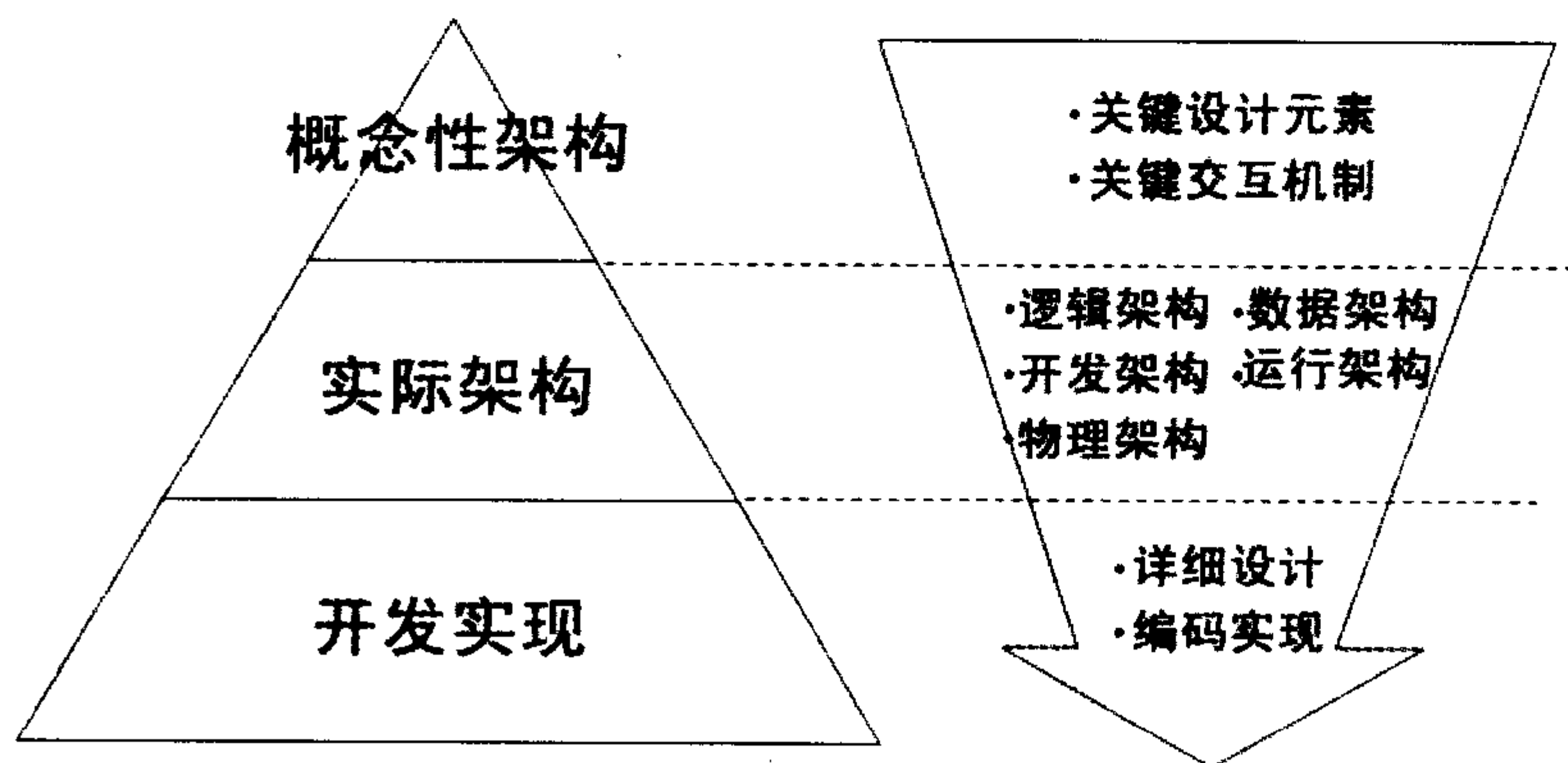


图 16-1 概念性架构、实际架构、开发实现的关系

### 16.1.2 运用基于 5 视图方法进行架构细化

为了基于 5 视图方法进行架构细化，我们需要运用多方面的知识、经验和上游工作成果（如图 16-2 所示）：

- 关键需求是对软件架构设计起关键作用的需求子集，包含功能需求、质量（属性）需求、商业需求三类，架构细化必须注意满足这些需求；
- 领域模型是以面向对象方式对问题领域的模拟和抽象，它揭示了重要的业务领域概念，并建立业务领域概念之间的关系；领域模型被不断精化之后成为最终软件系统的问题领域层，它决定了软件系统的功能范围，并影响着软件系统的可扩展性；
- 概念性架构是对系统设计的最初构想，通过主要的设计元素及它们之间的关系来描述系统，这些高层次的设计选择对未来软件系统的质量和功能都起着关键影响；
- 约束可以视为一类特殊的需求，它们具有强制性，规定了业务和技术上的标准和限制；另外，正如第 10 章所述，约束往往还会“衍生”出新的功能或质量属性要求，因此分析和设计时必须注意；
- 最后，软件架构师必须运用自己的经验和业界的经验。

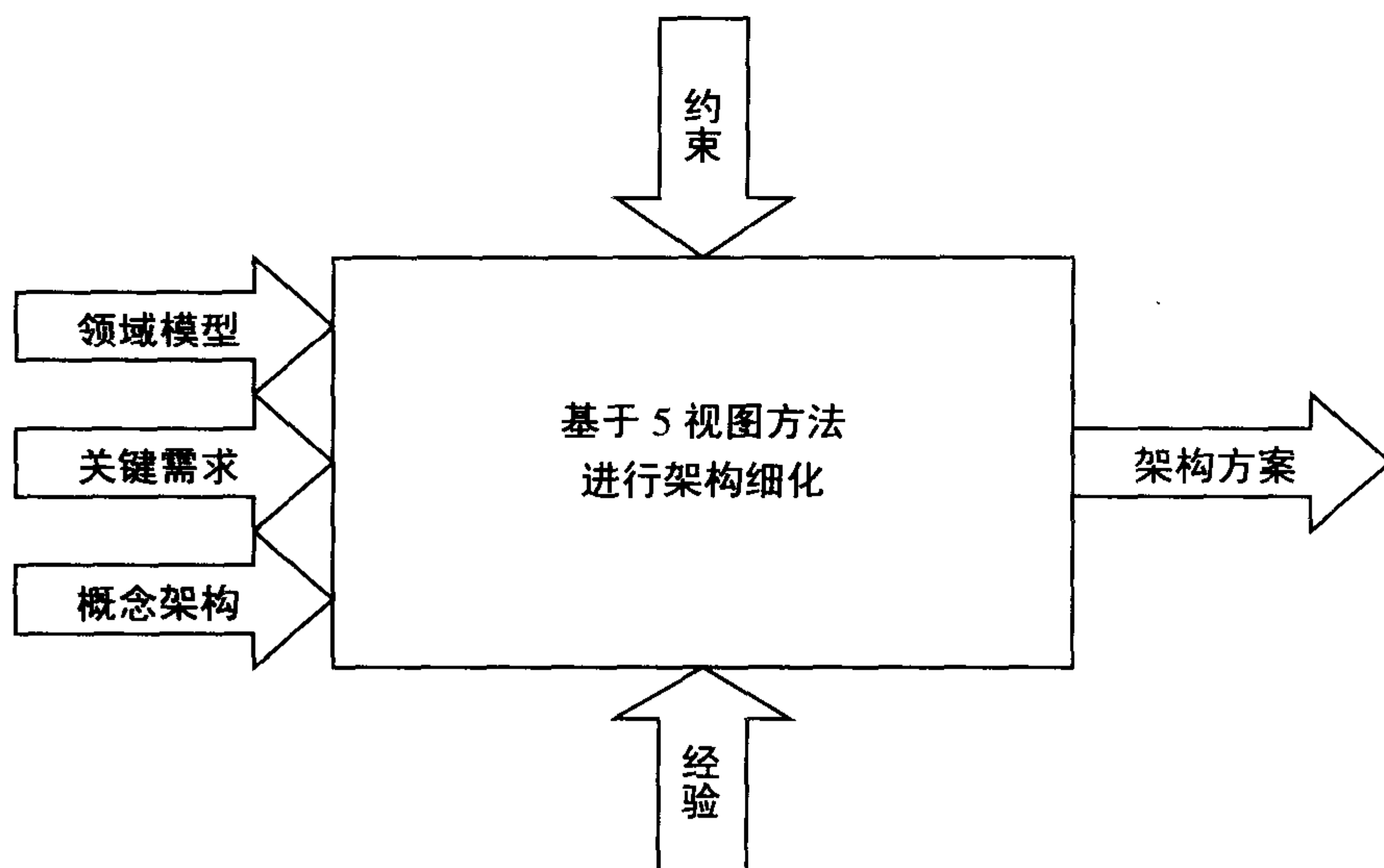


图 16-2 架构细化的“输入”和“输出”

这里有一个问题：5 视图方法中不同方面的设计有严格的先后顺序吗？

正如 Grady Booch 在《面向对象分析与设计》中所强调的：“每个项目都是很独特的，因此开发人员必须努力保持微观过程的非正式性和宏观过程的正式性之间的平衡。”同样，架构设计

细化活动中，5 个架构视图的设计顺序有很高的灵活性。换句话说，对于先设计哪个架构视图并没有硬性规定，反倒是不断转换架构设计的角度更能反映实际情况。例如，逻辑架构可能是不少人最早细化的架构视图，但有些分布式应用却从物理架构开始细化比较好（比如在客户组织本身就要求“分布”的情况下）。

## 16.2 设计逻辑架构

### 16.2.1 概述

逻辑架构的设计着重考虑功能需求——系统应当向用户提供什么样的服务。逻辑架构的关注点主要是行为或职责的划分，这不仅应包括用户可见的功能，还应包括为实现用户功能而必须提供的“辅助功能模块”，最终，将不同的职责分配给逻辑层、功能模块、类等不同粒度的逻辑单元。

如果使用 UML 来描述系统的逻辑架构，则该视图的静态方面由包图、类图、对象图来描述，动态方面由序列图、协作图、状态图和活动图来描述。

一般而言，逻辑架构的设计应完成下列工作：

- 细化功能单元
- 发现通用机制
- 细化领域模型
- 确定子系统接口和交互机制

### 16.2.2 识别通用机制

可见，逻辑架构设计中需要做很多工作。但其中的“发现通用机制”是应被特别强调的。我们说，软件架构关心的是如何将系统分为不同部分以及各部分之间如何交互，因此架构师在将系统划分为不同逻辑单元之后，必须考虑这些逻辑单元之间又是如何协作的。对此，可以有两种做法：

- 只识别并列举协作
- 识别协作，并将有共性的协作抽象成通用机制

图 16-3 说明了“只识别协作，不提取通用机制”的情况。不难理解，这会使最终的设计十分复杂，难以真正体现软件架构作为“新的设计层次”的作用。

其实，既然软件架构的重点在于“软件系统的各部分是如何相关的”，那么我们可以经过适度地抽象分析，将几组协作中的公共行为提取出来成为“通用机制”，这样有利于建立起所有涉众对软件架构的共同认识——即提高了系统的概念完整性。图 16-4 说明了这一点。



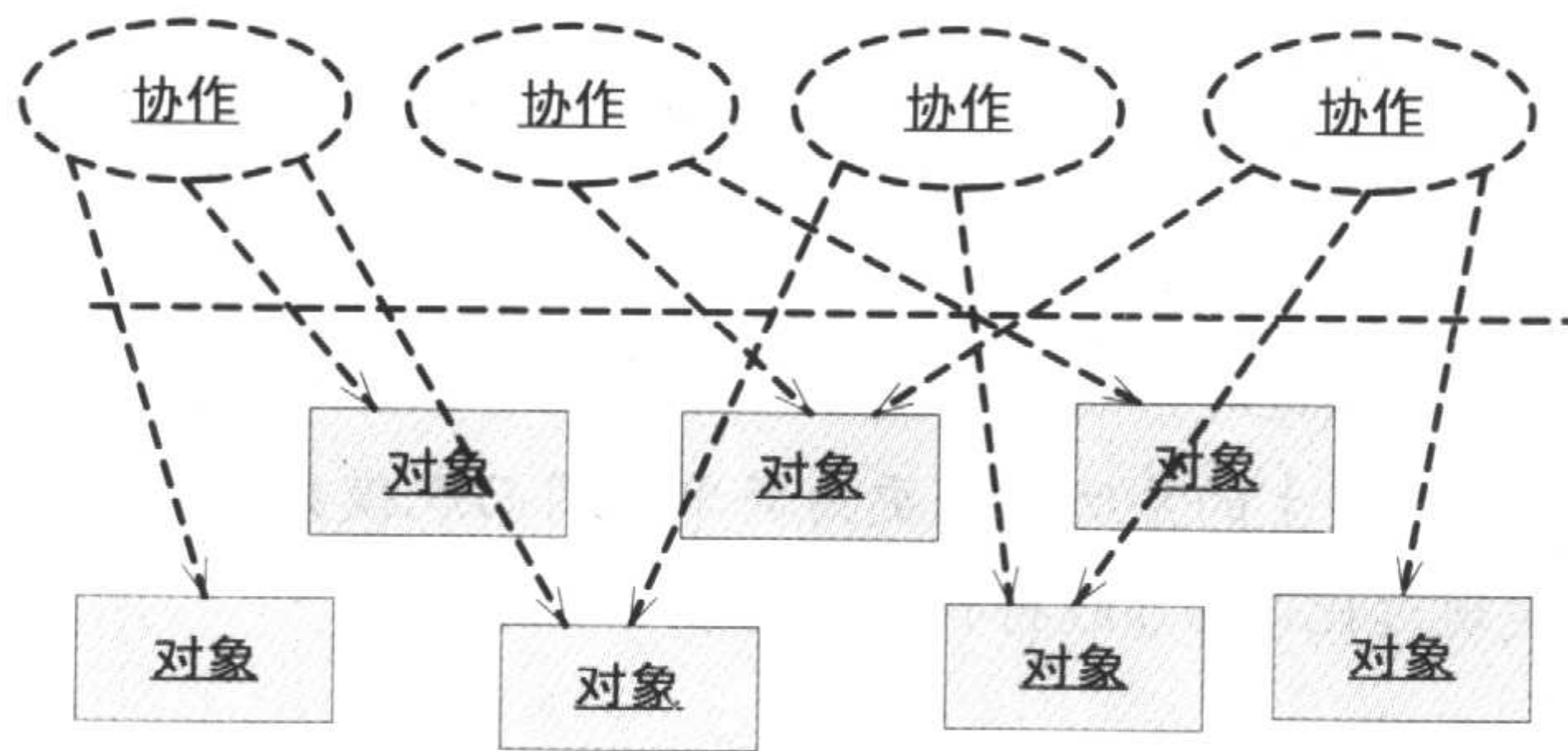


图 16-3 只识别协作，不提取通用机制

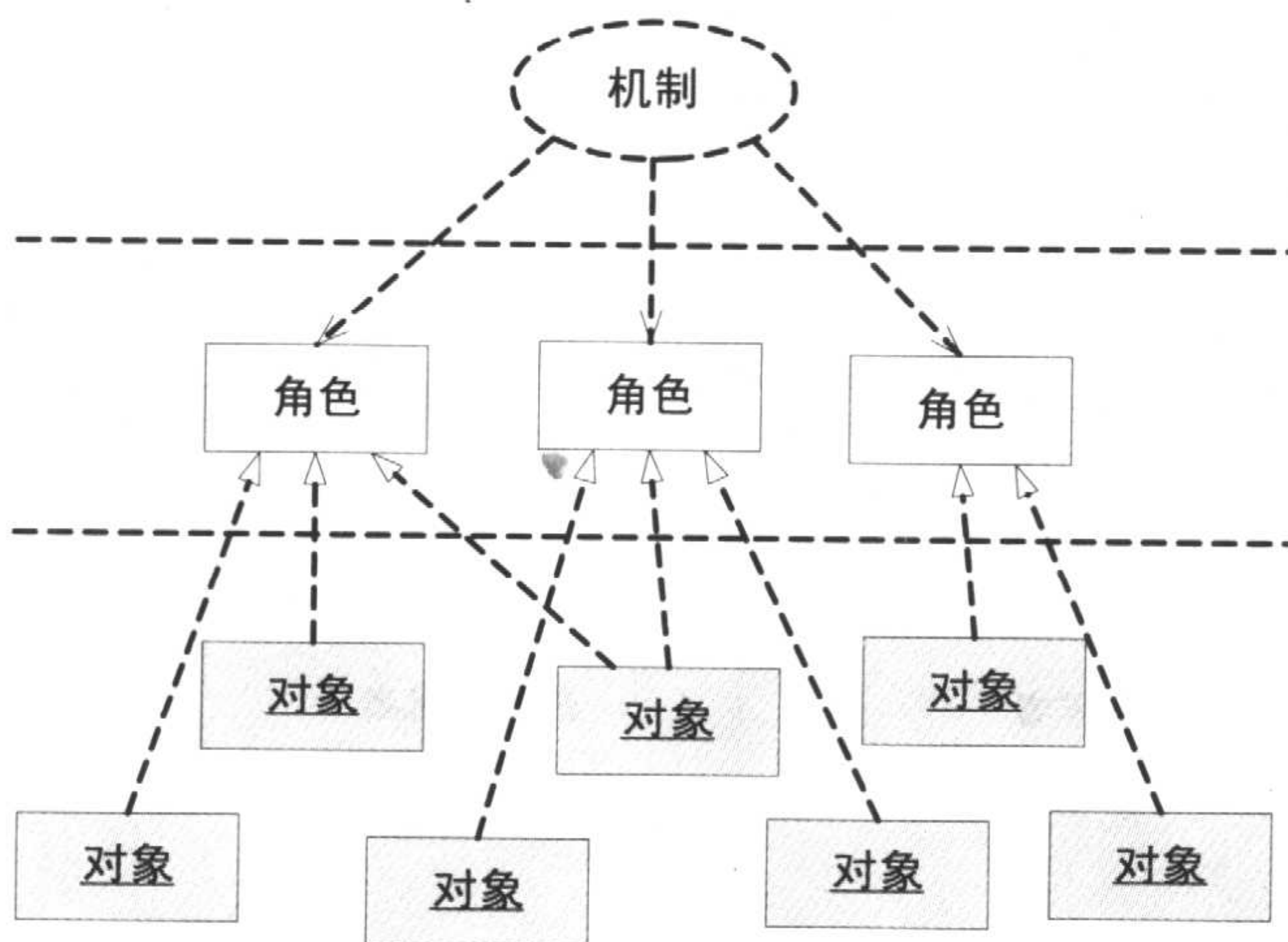


图 16-4 通用机制的提取

所谓机制，RUP 给出的定义是这样的：

机制（Mechanism）是模式的实例。机制必须进一步细化才能成为特定模型中的协作，因此，机制是独特上下文中重复出现的问题的特定解决方案。可以说，机制符合模式的定义。任何协作都可以被称为机制，但通常，机制仅指表述“软件应用系统中重复出现的问题的解决方案”的协作，例如可以采用模式的持久化处理等等（A mechanism is an instance of a pattern . It may require some further refinement to become a collaboration in a particular model. A mechanism is thus a specific solution , to a recurring problem, in a single context. A mechanism can be said to fit or conform to a pattern. Any collaboration could be termed a mechanism, but the term is usually reserved for collaborations which deliver a solution to a commonly recurring problem in software applications, for example, to handle persistence, to which a pattern is applicable. ）。

不少软件大师都很重视通用机制的提取，例如 Grady Booch 曾在《面向对象项目的解决方案》中强调了通用机制对架构设计的重要性，并谈到了概念完整性问题（《人月神话》中“概念完整性是产品质量的核心”的观点被广泛认同）：

具有良好架构的系统具备概念完整性。它通过对系统架构建立一种清晰的认识来发现通用的抽象和机制。利用这种共性使最终产生的系统结构更为简单，因而规模更小且更可靠。一个系统用 10 000 行代码实现远比 100 000 行代码好得多。

.....

一流是每个程序设计人员向往并为之奋斗却又无法具体说出的、难以达到的境界。一流的软件非常简明。它灵活而清晰，能通过创造性的机制解决复杂的问题，这些机制语义丰富，可应用于其他可能完全无关的问题。一流意味着寻求恰当的抽象，意味着通过新的途径合理利用有限的资源。

.....

记住，类很少单独存在。因此，尤其当考虑到系统的动态性时，应重点考虑某几组对象如何协作以便使用通用机制来处理公共行为。

下面举个例子。图 16-5 展示了一个采用对象图表达的“服务请求队列机制”。

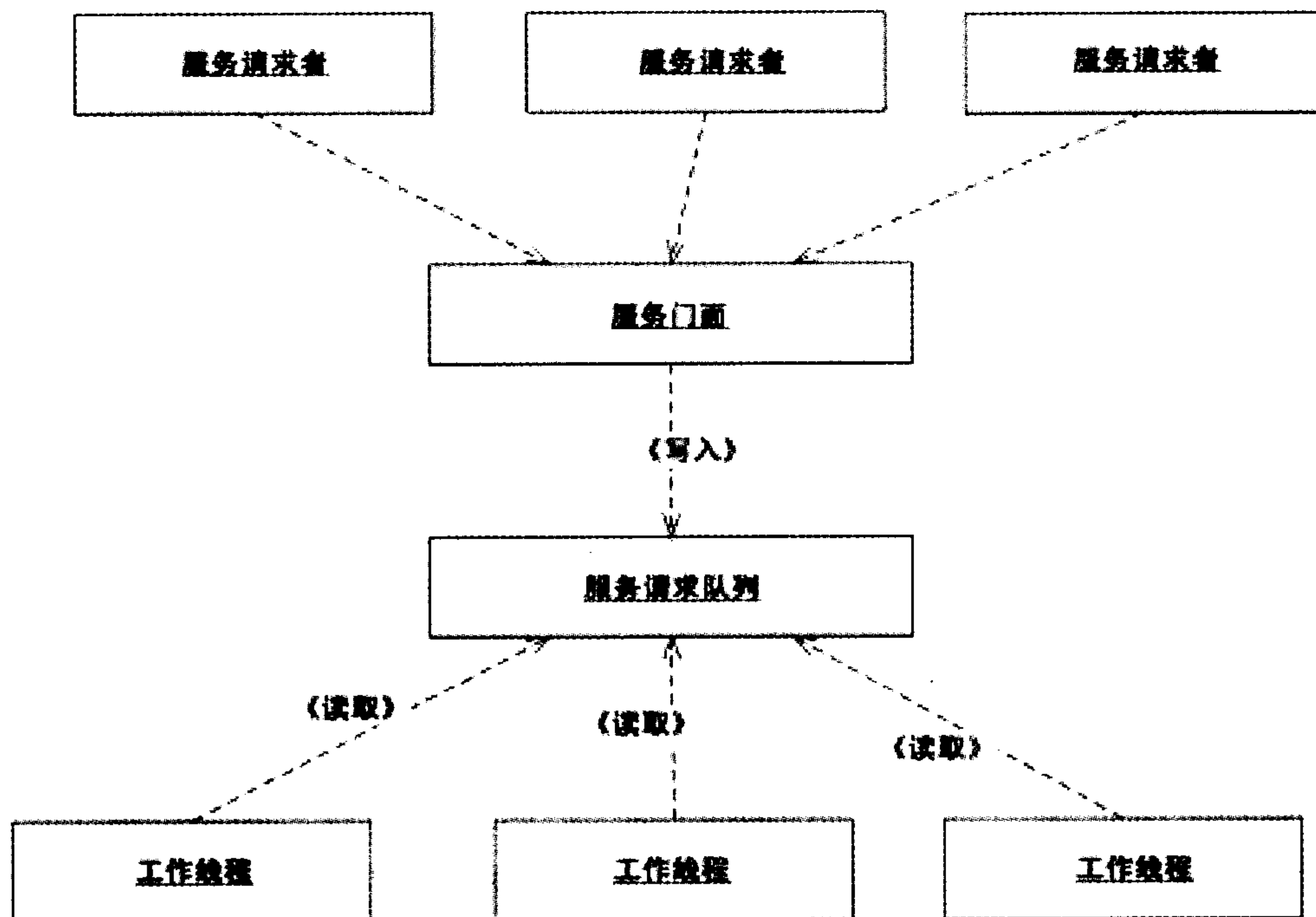


图 16-5 采用对象图表达的“服务请求队列机制”

## 16.3 设计开发架构

### 16.3.1 概述

开发架构的设计着重考虑开发期质量属性，例如可扩展性、可重用性、可移植性、易理解性、易测试性等。开发架构的关注点是在软件开发环境中软件模块的实际组织方式，具体涉及源程序文件、配置文件、源程序包、编译（或许还需要打包）后的目标文件、第三方库文件等。

有意思的是，开发架构和逻辑架构什么关系？答案是开发架构和逻辑架构之间存在一定的映射关系：比如逻辑架构中的逻辑层一般会映射到开发架构中的多个程序包；再比如开发架构中的源码文件可以包含逻辑架构中的一到多个类（在 C++ 里一个源码文件可以包含多个类，即使在 Java 里一个源码文件也可以同时包含一个类和几个内部类）。必须指出，当软件系统规模比较小时，这种界限可能很模糊。例如，将逻辑包和程序包等同起来直接规划程序包，以及将类和源程序文件等同起来直接规划类文件，这样做很适合较小的项目。

如果使用 UML 来描述架构的逻辑架构，则该视图可能包括包图、类图、组件图等。例如，图 16-6 展示了采用组件图表达的编译关系的一个例子。

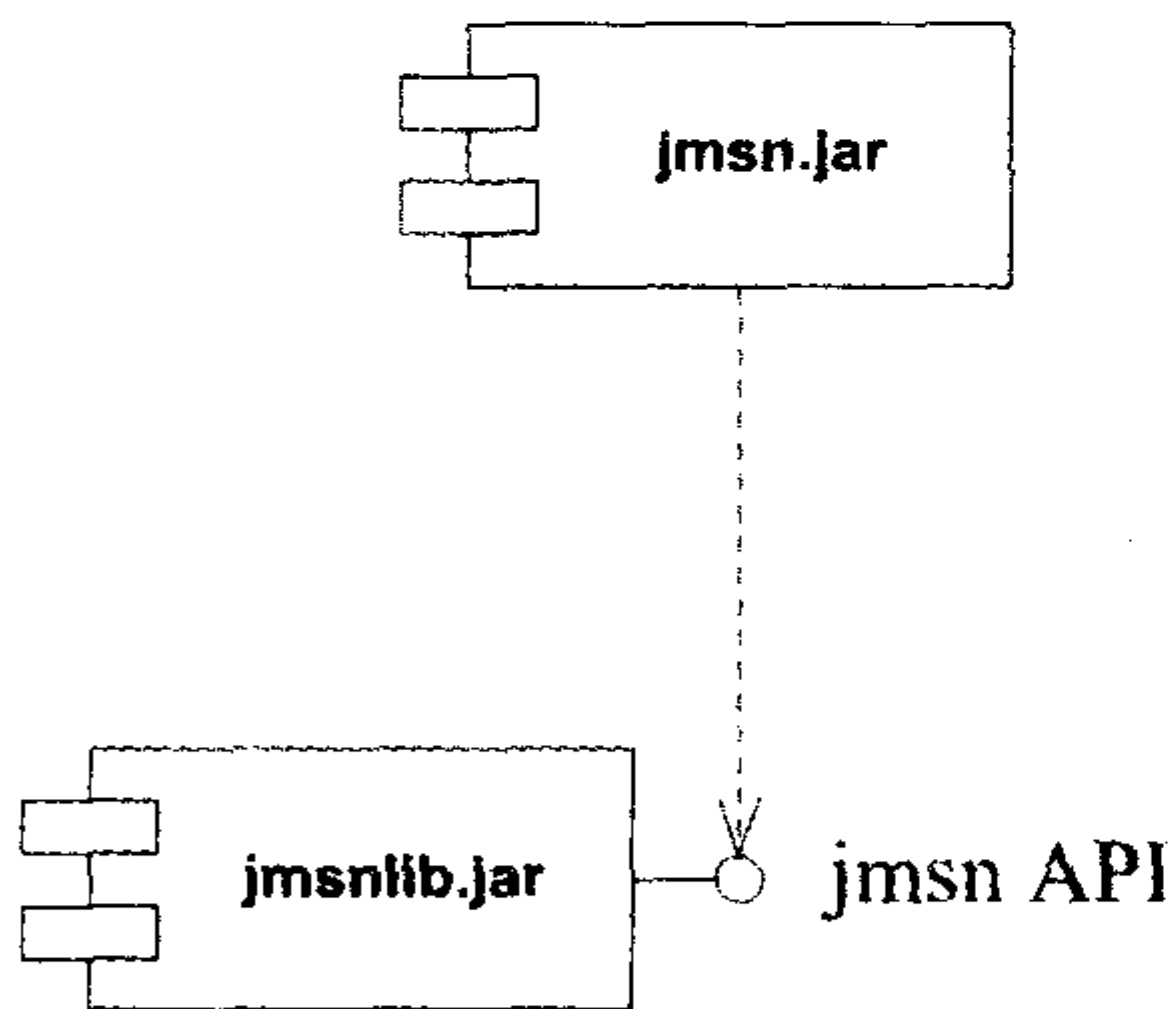


图 16-6 采用组件图表达的编译关系

一般而言，开发架构的设计应完成下列工作：

- 确定要开发或直接利用的程序包之间的依赖关系
- 确定采用的技术
- 确定采用的框架等

### 16.3.2 分层和分区

变化无处不在。好的架构设计必须把变化点错落有致地封装起来。

分层架构模式为“把变化点封装起来”提供了手段。分层架构的最大优点是将整体问题局部化，把可能的变化分别封装在不同的层中。最终，系统被规划为一个单向依赖的分层体系，利于修改、扩展、替换。从“行政”角度来讲，将架构划分为层的一个很大好处是，这些层形成了开发小组的自然分界——每层的开发人员所需要的技巧是不同的。例如，用户界面层的开发小组需要了解将使用的用户界面工具包；数据管理层的开发小组需要熟悉相关的数据库、持久工具或者使用的文件系统……

但分层并不足够，仅举一例。还记得第 4 章和第 5 章的案例吗？那是个设备调试系统。图 16-7 试图说明设备调试系统的分层架构，但并不理想。因为我们会产生如下一些困惑：

- 应用层将借助 MFC 实现，但图中没有反映清楚；
- 通讯层的实现基于某串口通讯 SDK，图中也没有反映清楚；
- 基础框架层为什么要放在设备控制层之下呢？难道设备控制层作为嵌入式应用，也会“使用” Windows 桌面应用的框架 MFC 吗？看来有点儿歧义；
- 层次划分的依据不一致。如果说根据稳定性大小不同，把最稳定的基础框架层放在最下面，那么设备控制层在通讯层之下却不能说设备控制层更稳定。如果说应用层、通讯层、设备控制层是根据职责不同而进行的“高内聚”的划分，那么基础框架层的职责怎么说都不“高内聚”。

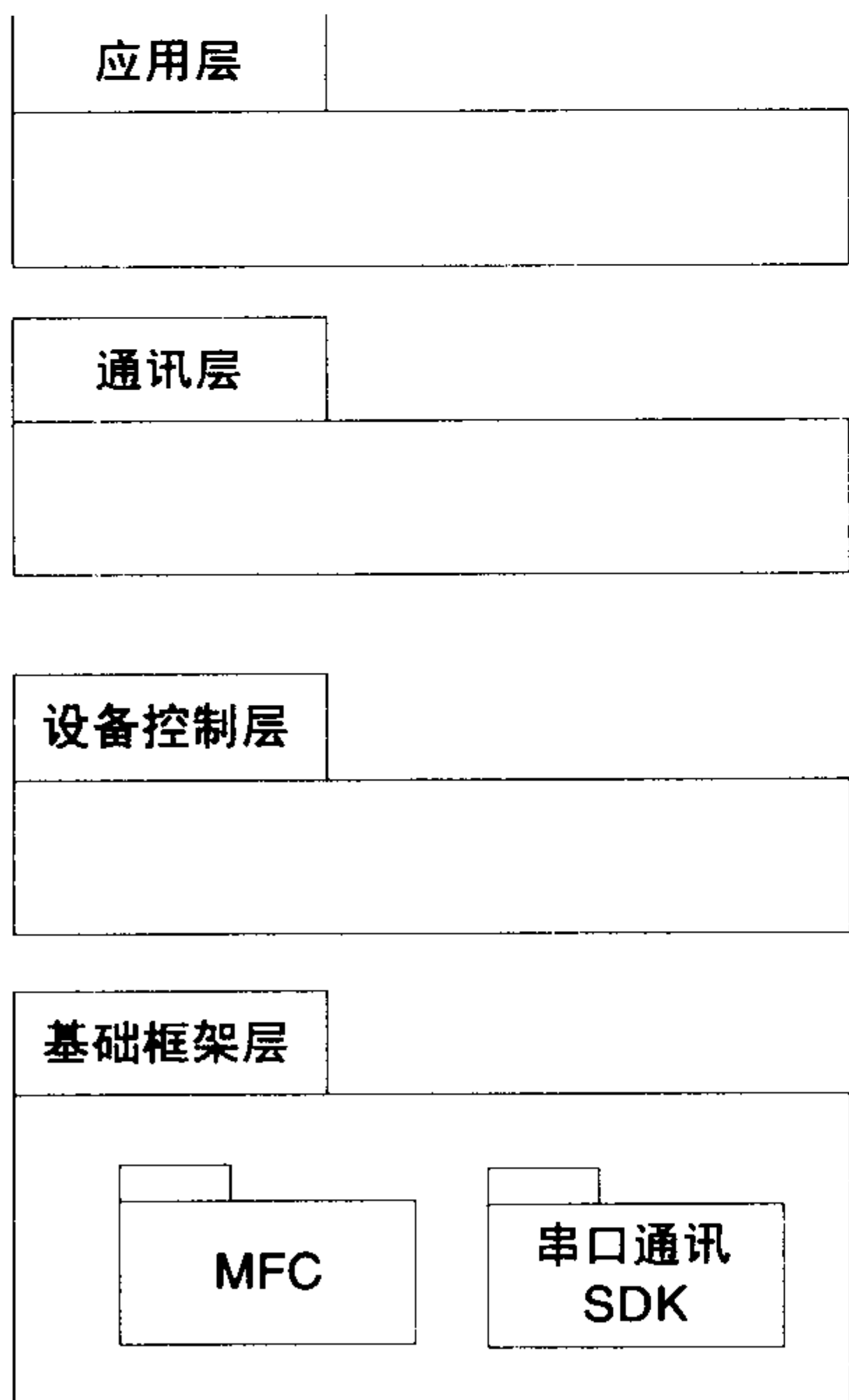


图 16-7 令框架位于“基础框架层”



于是我们想，既然开发架构中既涉及了需要编写的源程序，又包括了第三方的框架，那么我们必须将它们的关系表达出来。将各种框架集中到“基础框架层”放在最下面并不总是合适，因为大量框架属于白盒框架的范畴，无论这些框架是第三方开发的、还是团队自己抽象出来的，它们都应该“待在”原来的层。为此，我们应该在分层的基础上，进一步分区（Partition），形成横切竖割的“二维架构”。图 16-8 展示了新的设计图。

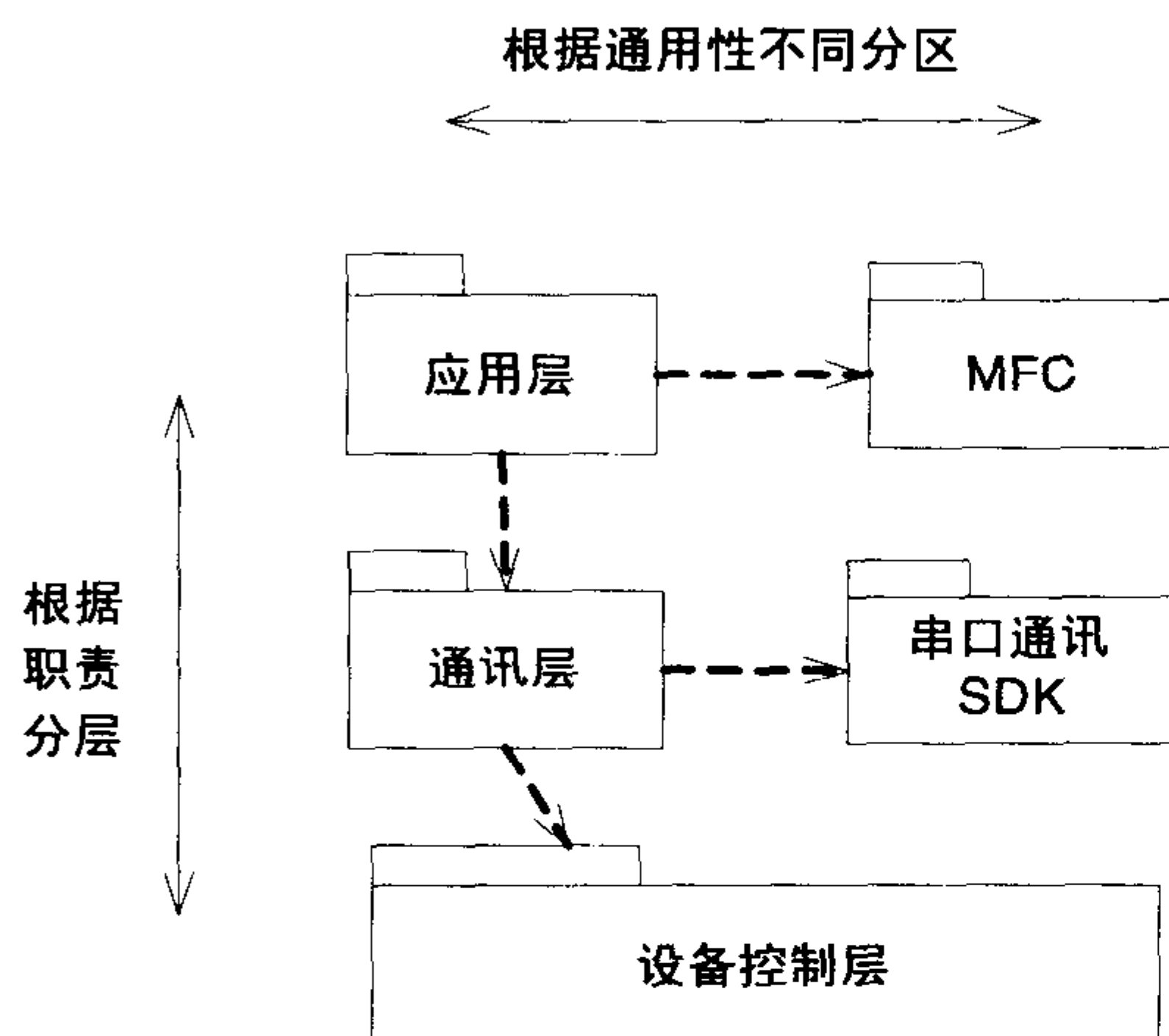


图 16-8 通过分区来说明框架在架构中的位置

更进一步，我们还可以把类库和框架的具体用法反映出来。特别是在框架实现了关键的架构机制的情况下，这是很必要的。图 16-9 所示为 JGraph 框架的一个应用案例（JGraph 是一个开源的基于 MVC 架构的图形组件）。我们知道，框架封装了设计元素之间的交互机制和协作流程，无论框架是自开发的还是第三方实现的，这些设计思想都是架构的一部分。而分区很适合表达应用使用框架的方式：通过类继承或接口实现等手段对扩展点进行扩展。

总之，现在我们的开发越来越多地依赖于现成框架，能否切中肯綮地阐明开发和框架的关系，不仅关系到你的设计思路是否清晰、是否合理，还关系到后续的程序开发工作能否顺畅展开的问题。同时，软件架构为了达到易修改、易测试、易扩展等质量属性需求，必须分离关注点，而在通过展现层、业务层、数据层等进行关注点分离的同时，可以辅以分区方法。框架是分区方法最常用的例子，可以更进一步地分离关注点——将应用相关和应用无关的部分分离——将不同变化进行分门别类地封装。

最后需要补充说明的是，分区其实也适用于“逻辑架构的设计”，我们之所以将分区方法放在“开发架构的设计”中讲，只不过将分区用于“找准框架在架构中的位置”最为典型罢了。

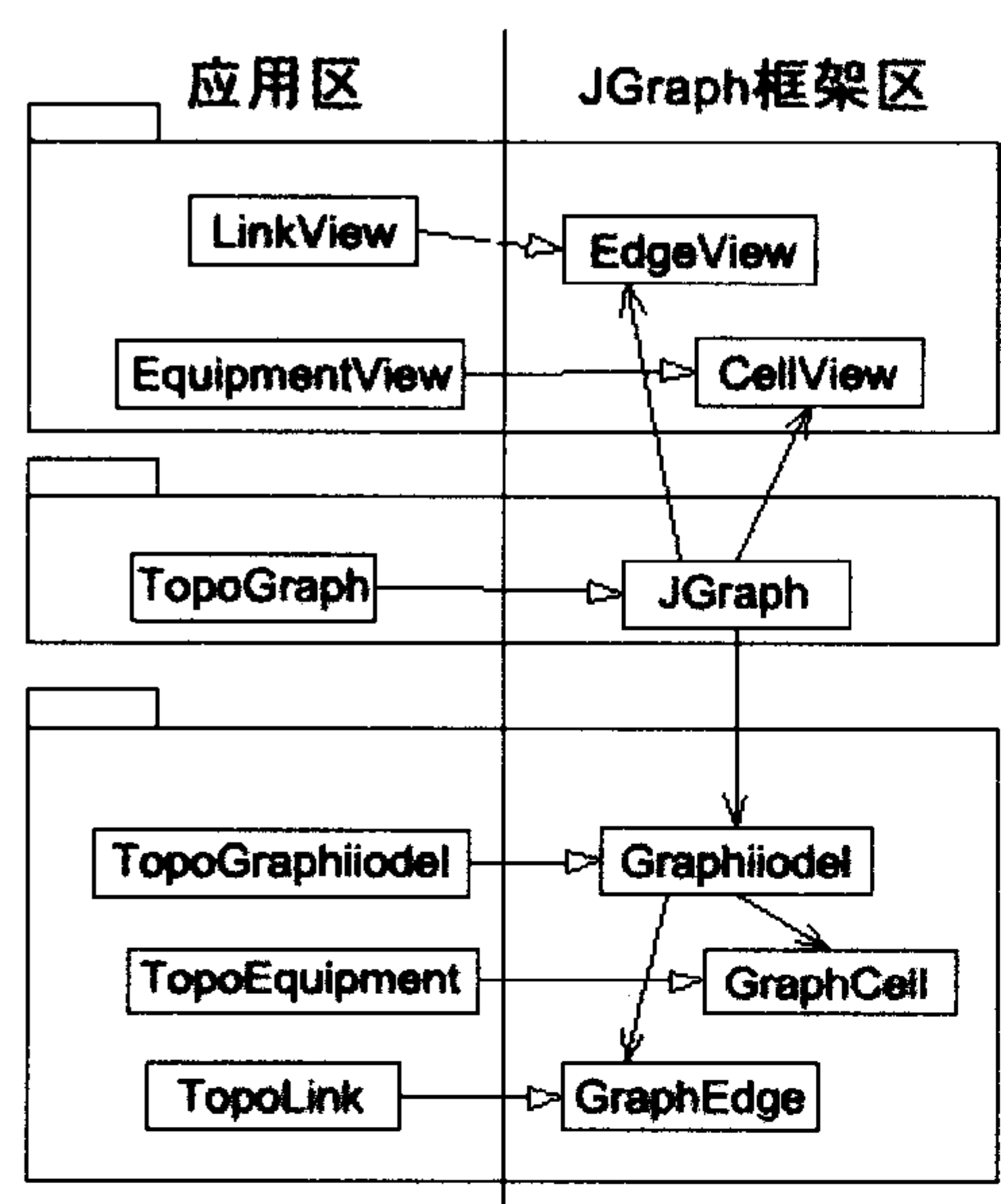


图 16-9 开发架构设计：运用分区反映框架的用法

## 16.4 设计数据架构

### 16.4.1 概述

数据架构的设计着重考虑“数据需求”。有人说，系统=程序+数据+硬件，可见数据的重要性。有时候，程序升级了，硬件更新了，但数据不能丢——要么保留，要么移行。因此数据的规划应该成为架构设计的重要组成部分。

数据架构的关注点是持久化数据的组织。对于很多集成系统，数据需要在不同系统之间传递、复制、暂存，这往往还会涉及到不同的物理机器；也就是说，如果需要，也可以把数据放在物理架构之中考虑，以便体现集成系统的数据分布与传递特征。

数据架构的描述一般采用 E-R 图和数据流图表示，当采用 UML 时，可以用 UML 类图替代 E-R 图，采用带对象流的活动图替代数据流图。

一般而言，数据架构的设计应完成下列工作：

- 持久化数据存储方案
- 数据传递、数据复制、数据同步等策略（可选）

### 16.4.2 如何将 OO 模型映射为数据模型

如图 16-10 所示，数据架构的设计可以有不同的方法：既可以从 OO 模型开始，也可以从经典的逻辑数据模型开始。本书推荐先基于 OO 模型进行物理数据模型的设计，之后进行设计优化。因为如果数据架构的设计忽视了领域模型，重新从逻辑数据模型开始，就人为地“切断”了面向对象模型和数据模型之间的联系，也不利于维护 OO 模型和数据模型之间的一致性。

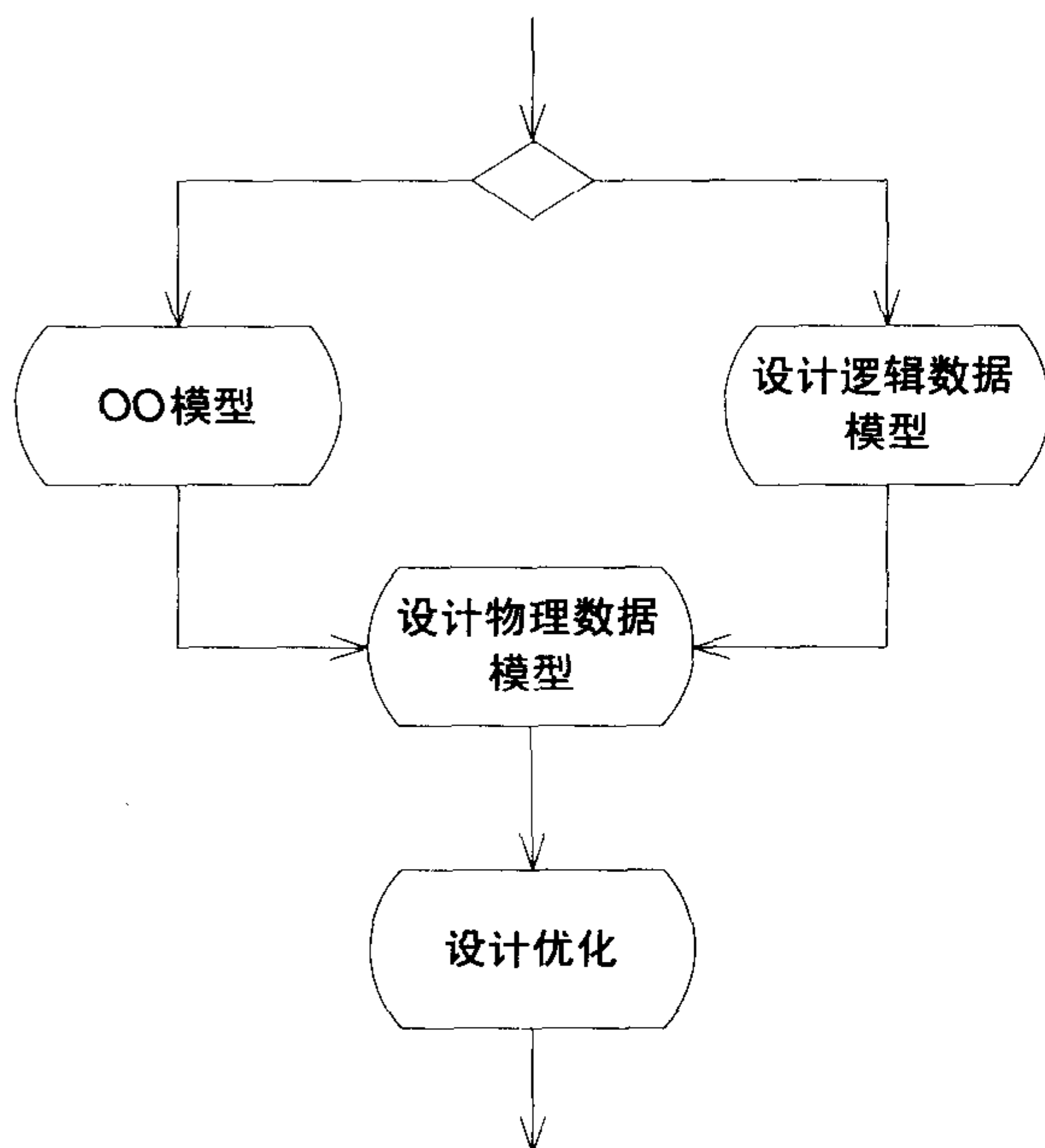


图 16-10 设计数据架构的步骤

从 OO 模型到数据模型的映射，有着完整的规则体系（ORM 工具就是借助了这些规则完成从对象模型到关系模型的转换）。下面通过例子，来说明 OO 模型和数据模型的紧密关系。图 16-11 展示了商品分类的类图和相应物理数据模型，其中规定，任何商品必须位于严格的二级分类体系之中。

如果我们将分类体系改为灵活的多级分类，则类图将被调整，物理数据模型也应相应被更改，如图 16-12 所示。

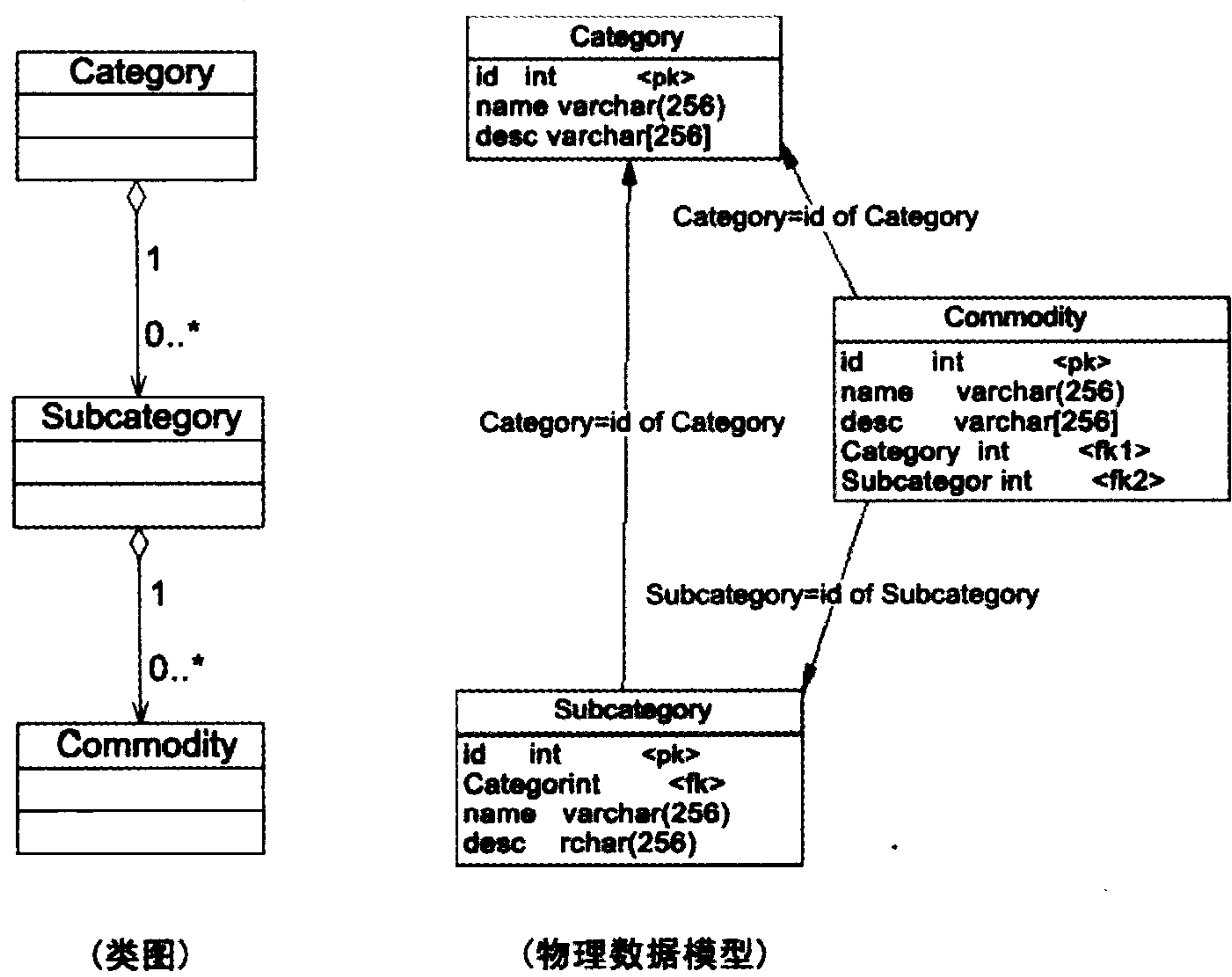


图 16-11 二级分类的商品之例

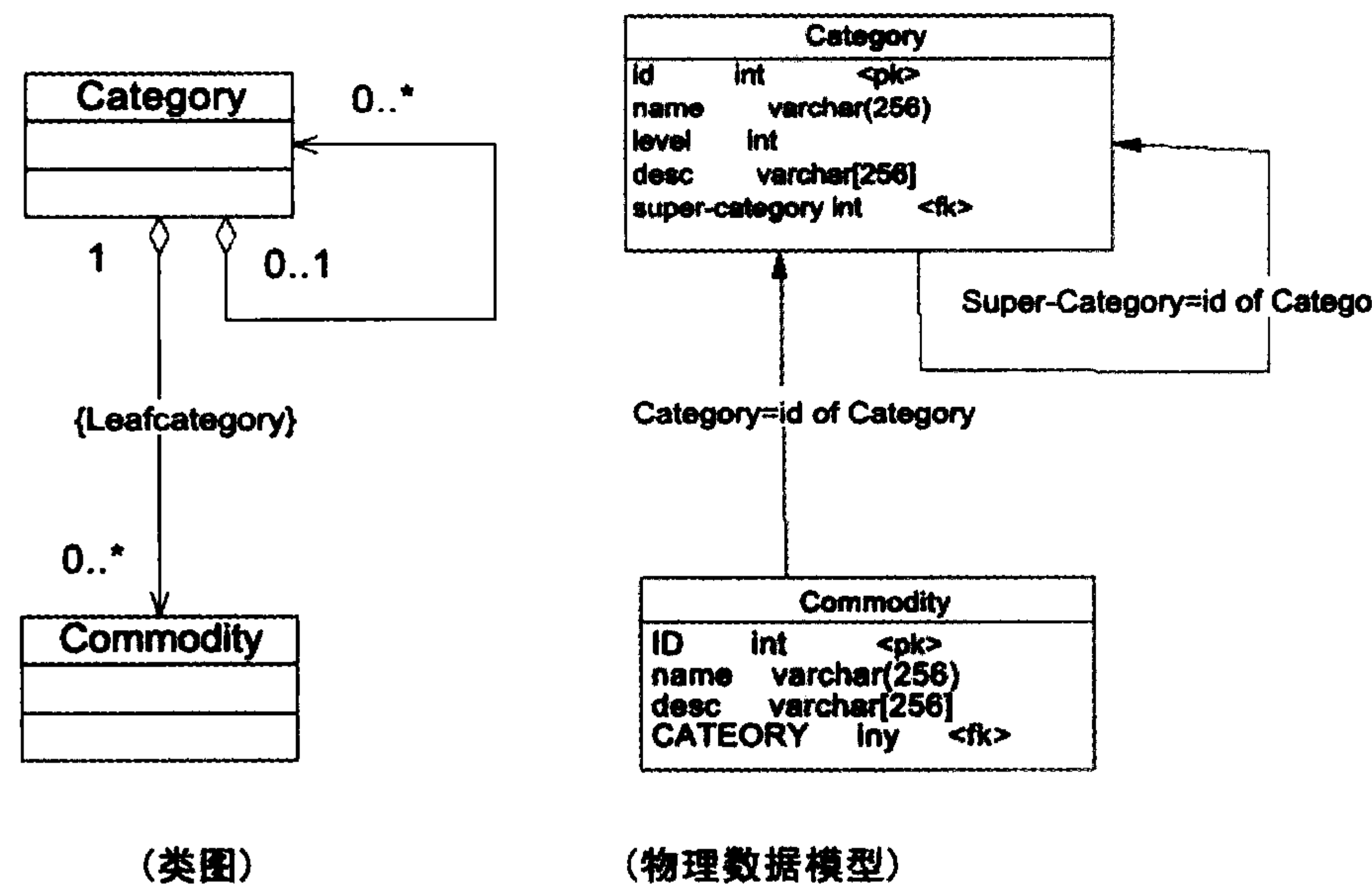


图 16-12 多级自由分类的商品之例



继续我们的例子。上面的两种分类方法都规定“每种商品只能被归在一种分类之中”，这显然对商品的检索和销售不利，例如保罗·莫里哀的唱片，既应该在“CD→轻音乐→外国轻音乐”分类中找到，也应该在“纪念品→保罗·莫里哀”分类中找到。图 16-13 展示了调整之后的 OO 设计和物理数据模型设计。

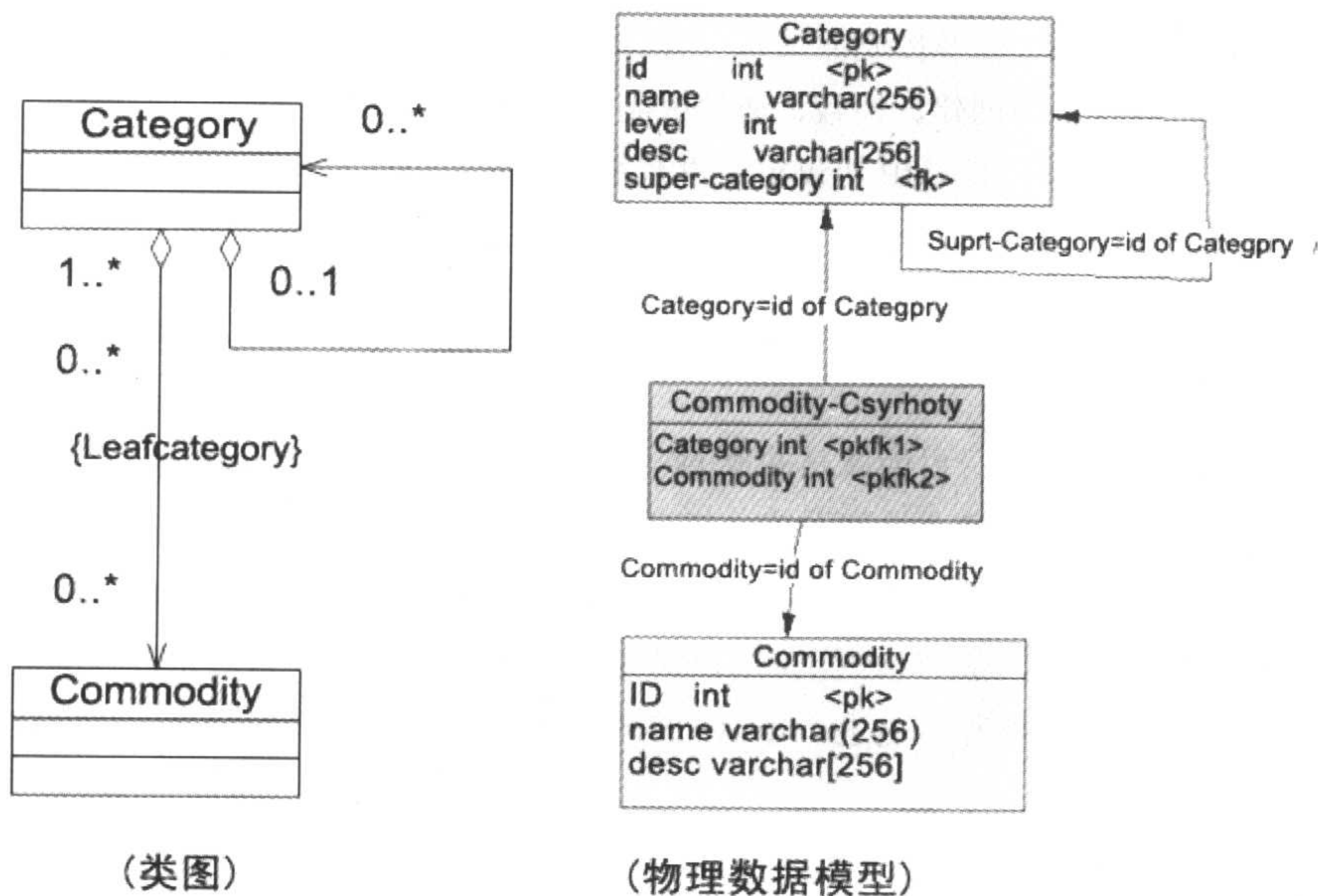


图 16-13 商品可属于多种类别之例

可见，OO 模型和数据模型之间关系密切，从 OO 模型导出数据模型的做法可以更好地保持设计的一致性。

## 16.5 设计运行架构

### 16.5.1 概述

运行架构的设计着重考虑运行期质量属性，例如性能、可伸缩性、持续可用性等。运行架构关注进程、线程、对象等运行时概念，以及相关的并发、同步、通信等问题。

运行架构和开发架构的关系非常值得研究。开发架构一般偏重程序包在编译时期的静态依赖关系，而这些程序运行起来之后会表现为对象、线程、进程，运行架构比较关注的是这些运行时单元的交互问题。总之，运行架构是在开发架构的基础上，从宏观上规划多条控制流的并发和同步。

如果使用 UML 来描述架构的运行架构，则该视图的静态方面由包图、类图（其中主动类非

常重要)、对象图(其中主动对象非常重要)等来说明关键运行时概念的结构关系,动态方面由序列图、协作图等来说明关键交互机制。

一般而言,运行架构的设计应完成下列工作:

- 确定引入哪些进程与线程
- 确定主动对象、被动对象、以及控制流关系
- 处理相关问题:进程线程的创建、销毁、通信机制、资源争用等
- 协议设计(可选,例如基于 TCP/IP 协议定义本系统的“应用协议”)

## 16.5.2 运用主动类规划并发

为什么要进行并发性的设计呢?

一方面,软件系统要支持的业务本身可能就是并发的,例如业务中要求同时处理多种请求、同时为很多用户提供服务,或者同时和多个外部系统“打交道”等。

另一方面,并发利于提高性能和可伸缩性,充分利用计算机资源。例如,本书曾分析过的设备调试系统的案例(可参考图 5-5),为了同时保证用户对界面响应速度的要求和通讯数据的处理,为不同服务部分纳入互相独立的线程。更进一步地,还可以为某些控制流设置较高的执行优先级,进行更复杂的控制。

还有一种情况,那就是在某些情况下,采用并发性设计是最直观、最自然的解决方案。例如两个进程分别扮演“消息”的消费者和生产者,分别读写“消息队列”,这种方法常被用于软件系统服务器端的架构设计之中。

具体而言,控制流作为在处理机上顺序执行的动作系列,目前主要的实现技术有两种:进程或线程。进程被称为“重量级控制流”,因为它既是处理机资源的分配单位,又是其他计算机资源的分配单位。线程则被称为“轻量级控制流”,它仅仅是处理机资源的分配单位。一个进程内可以包含多个线程,这些线程共享所在进程的资源,在线程之间共享进程的数据是常用的设计技巧;但处理机资源例外,线程是独立的处理机资源的分配单位。

在面向对象方法中,线程也应通过对象来封装,这涉及到两个概念:主动对象和主动类。主动对象(Active Object)是一组属性和服务的封装体,其中至少有一个服务是主动服务,主动服务不需要接收“消息”就能主动执行。与主动对象相对的概念是被动对象(Passive Object),它的所有服务都是被动服务,需要被调用才能执行。主动类(Active Class)则是描述主动对象的类。

主动对象对于并发性设计之所以重要,是因为它是控制流的驱动者。主动对象的主动服务是控制流的源头,该主动服务被创建成进程或线程,从而它可以获得处理机资源并开始活动。从主动对象的主动服务开始,层层调用其他对象的服务,就形成一个控制流。



以 Java 语言为例，一个线程必须是由一个主动类定义的，并通过“专门机构”完成线程的启动。在 Java 中实现线程化有两种方式。第一种方式是实现为 Thread 类的子类，表 16-1 所示的代码为这种方式的一个具体例子，线程“乒乓”将不断地在屏幕上打印指定字符串。

表 16-1 代码

```
public class PingPong extends Thread { //此 PingPong 就是主动类
    private String word;
    private int delay;
    public PingPong(String word, int delay) {
        this.word = word;
        this.delay = delay;
    }
    public void run() {
        try {
            for (;;) {
                System.out.pring(word + " "); //此 out 对象可视为被动对象
                Thread.sleep(delay); //不是被动对象，是告知“专门机构”睡眠一会儿
            }
        } catch (InterruptedException e) {
            return; //函数 run()的出口，同时也是线程【结束】点
        }
    } //end of run
} //end of PingPong
```

上面的 Thread 子类 PingPong 可以这样被启动：

```
new PingPong("ping", 33).start(); //【创建】和【启动】
new PingPong("Pong", 100).start(); //将 run()创建成 OS 能感知的 thread
```

图 16-14 反映了方式一的结构。不难理解，Thread 类的背后“藏着”真正的线程机构，它最终将调用本地操作系统的能力来实现线程的创建与控制。

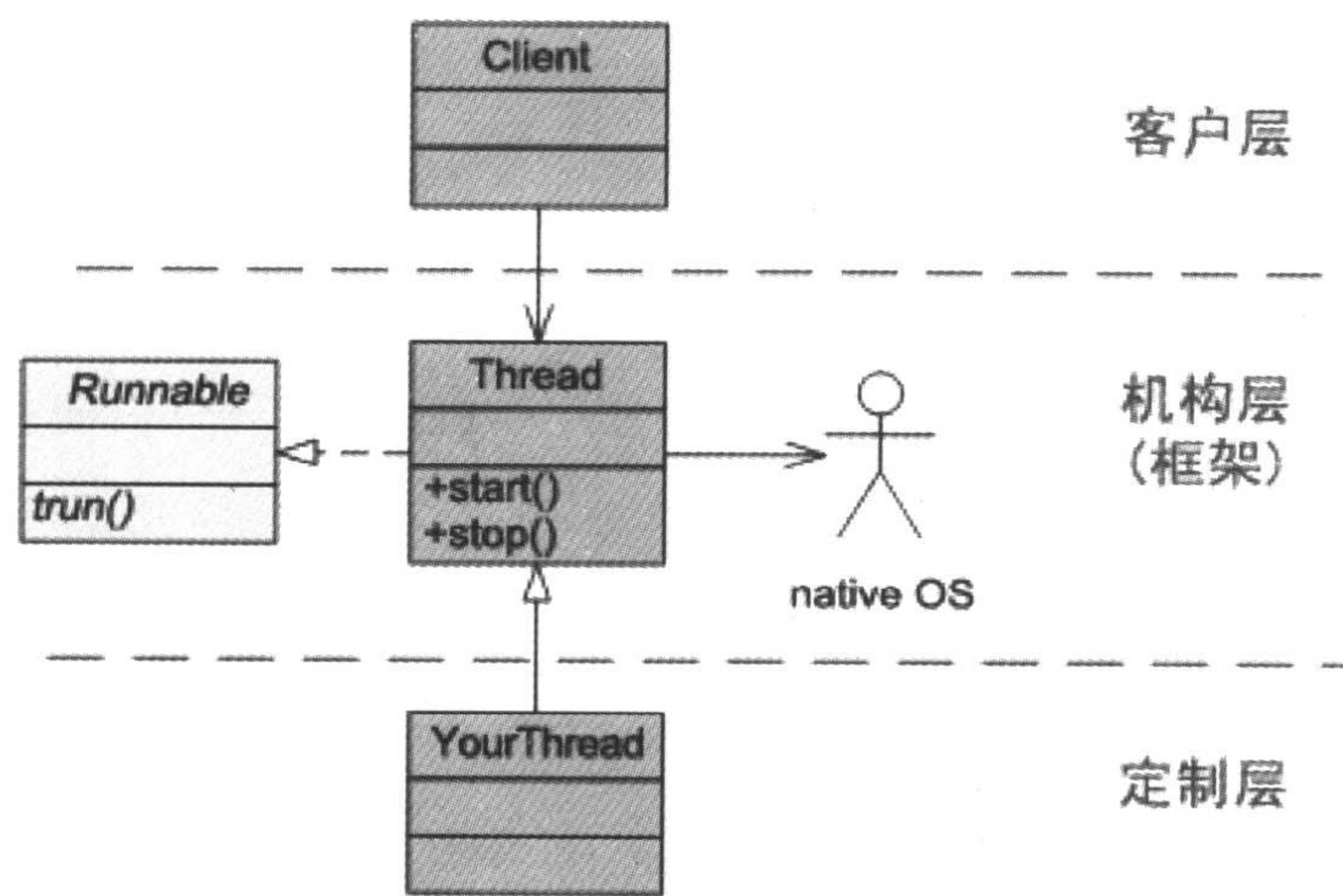


图 16-14 Java 实现线程的方式一：实现为 Thread 子类

表 16-2 的代码展示了 Java 实现线程的另一种方式：实现 Runnable 接口，并通过 Thread 最终将之创建成线程。

表 16-2 代码

```
public class PingPong { //此类为被动类
    private String word;

    public PingPong(String word) {
        this.word = word;
    }
    public printWord(){
        System.out.pring(word + " "); //此被动类又调用另一个被动对象 out
    }
} //end of PingPong

public class RunnablePingPong implements Runnable { //此类为主动类
    private PingPong pingpong;
    private int delay;
    public RunnablePingPong (String word, int delay) {
        pingpong = new PingPong(word);
        this.delay = delay;
    }
    public void run() {
        try {
            for (;;) {
                pingpong.printWord(); //此 pingpong 对象应视为被动对象
                Thread.sleep(delay);
            }
        } catch (InterruptedException e) {
            return; /////////////// 【结束】
        }
    } //end of run
} //end of RunnablePingPong

下面将主动类 RunnablePingPong 启动：
Runnable ping = new RunnablePingPong("ping", 33); /////////////// 【创建】
Runnable pong = new RunnablePingPong("Pong", 100);
new Thread(ping).start(); /////////////// 【启动】
new Thread(pong).start();
```

图 16-15 反映了方式二的结构。



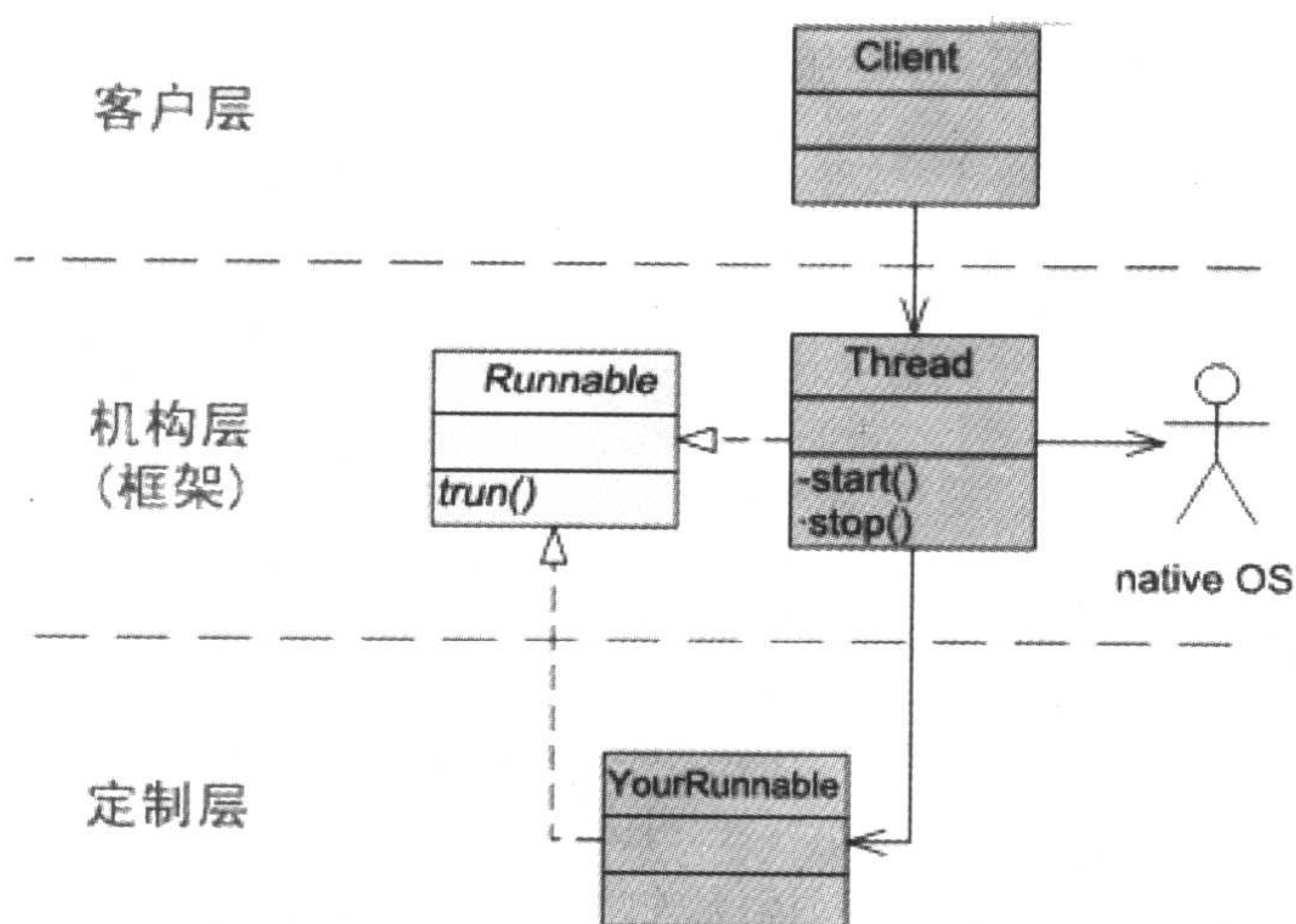


图 16-15 Java 实现线程的方式二：实现 Runnable 接口

结合方式二的例子，我们来理解“从主动对象的主动服务开始，层层调用其他对象的服务，就形成一个控制流”这句话。图 16-16 是例子二运行时的对象图，并明确标出了控制流。

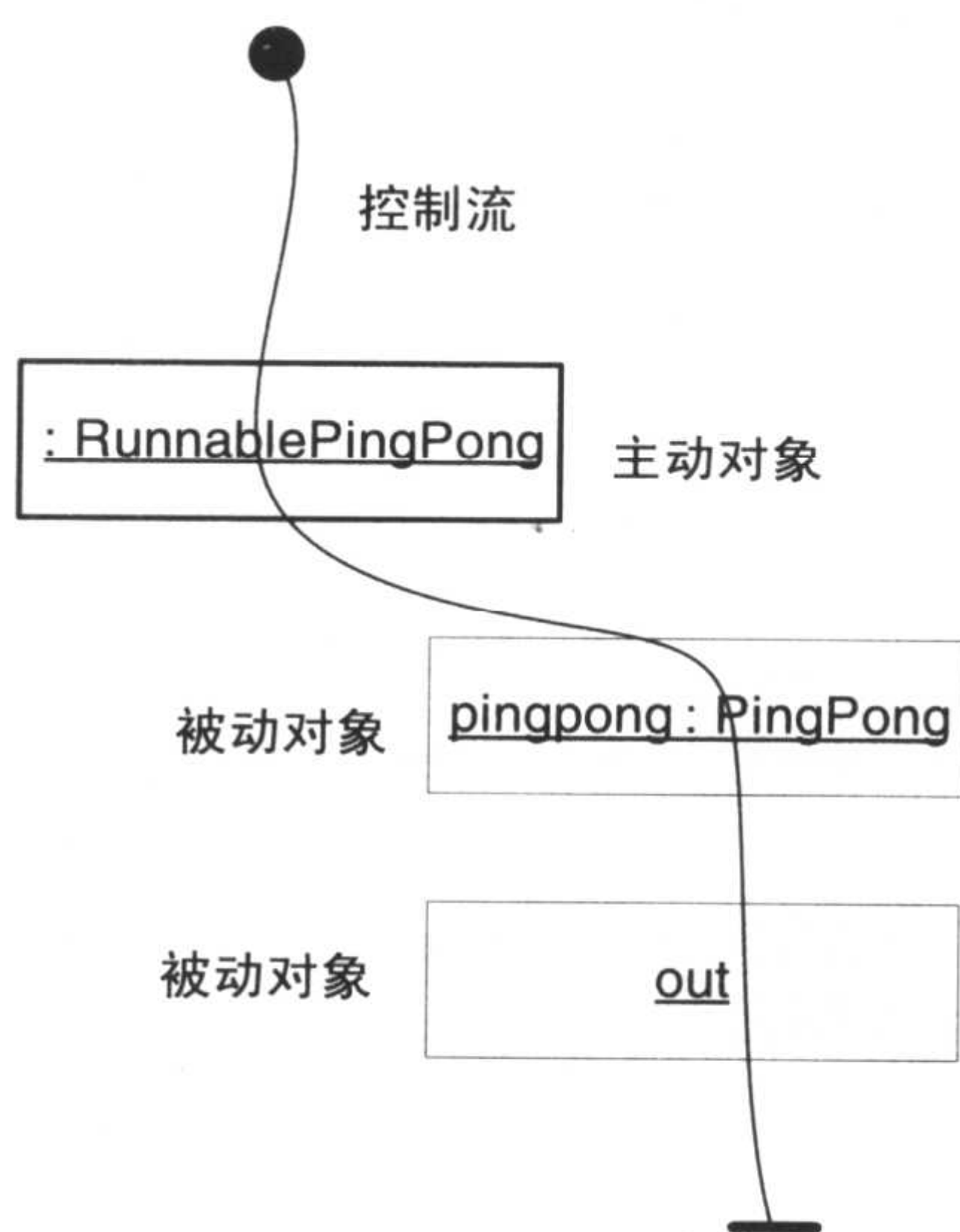


图 16-16 控制流串起主动对象和被动对象

上面的例子说明了如何运用 Java 语言实现控制流，其中主动对象是控制流的源头。其实，嵌入式软件系统的设计也可以借助主动对象来进行，只不过主动类的具体实现可能需要借助更多手段：一些用于嵌入式应用开发的平台（如 Kjava）直接支持线程，我们可以用 OOP 编写主动

类；对于不直接支持主动类的编程语言，借助编写“中断服务”来实现主动类的设计概念，这是一种非常重要的实现主动对象的手段。

总之，进行并发性设计的关键是识别系统中所有并发执行的控制流，然后用主动对象来启动这些控制流。把系统中所有的主动对象表示清楚，就抓住了系统中每个控制流的源头，就可以把并发执行的所有控制流梳理清楚。

16.5.3 应用协议的设计

为大多数应用系统设计架构时不会涉及应用协议的设计，但并不总是这样。

所谓应用协议，是指处于通信协议栈最高层的协议。例如，TCP/IP 协议栈分为四个层次：应用层、传输层、互联层、主机-网络层（如图 16-17 所示）。应用协议定义了网络应用程序之间是如何“交谈”的，它的语义是和应用相关的，而和具体通信无关。当然，应用协议还可以基于其他传输层协议之上来构建，并不一定是 TCP/IP 协议。例如，第 4 章的 4.5 节“设备调试系统案例：领会逻辑架构和物理架构”中我们曾基于 RS232 协议构建了系统所需的应用协议。

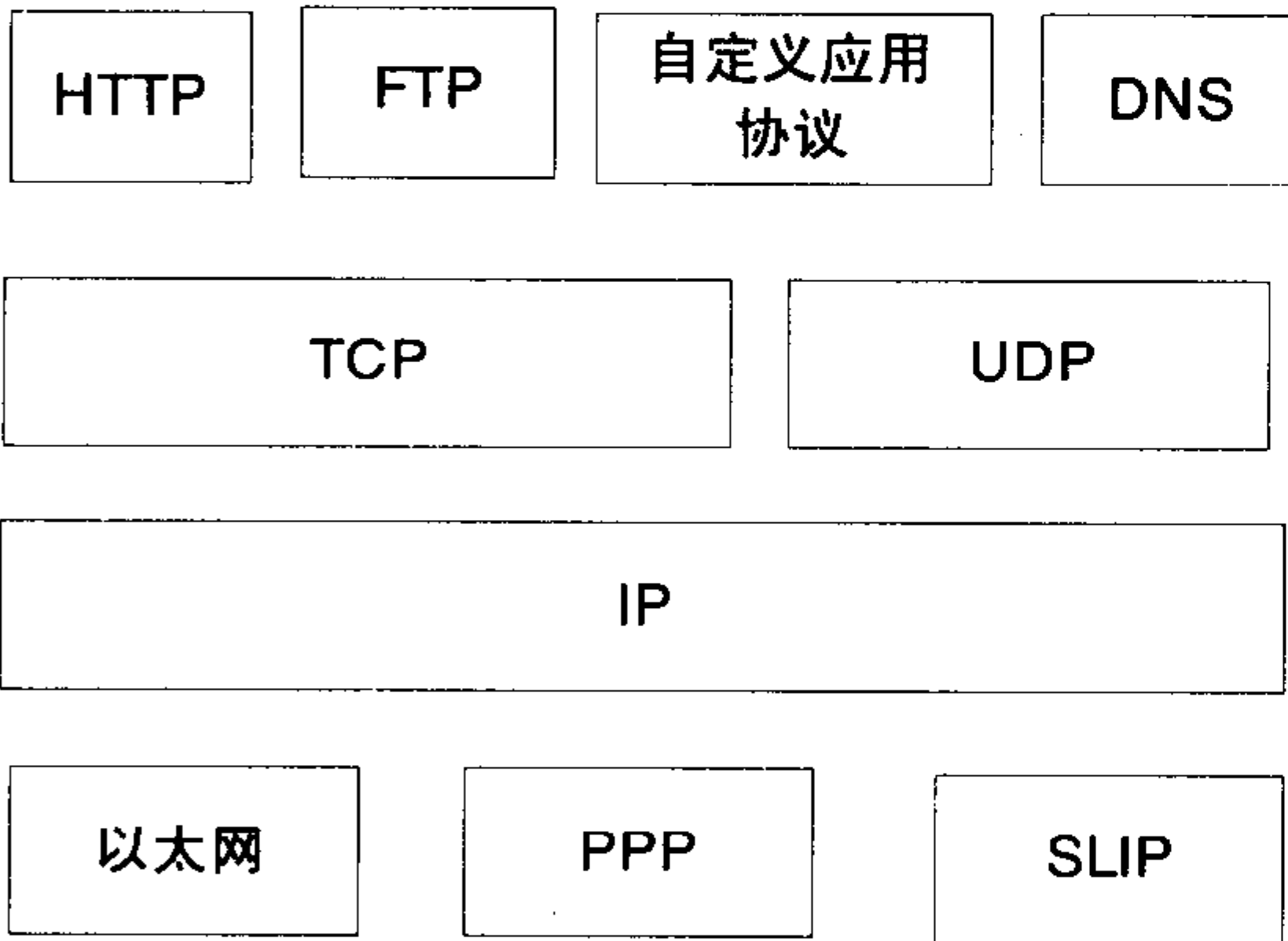


图 16-17 TCP/IP 协议栈中应用协议的位置

如果软件系统的设计涉及到了自定义应用协议的问题，那么它一定应该在架构设计阶段完成，而不能拖到详细设计阶段。一般而言，协议的设计属于运行架构设计应关注的范围。

16.6 设计物理架构

16.6.1 概述

物理架构的设计着重考虑“安装和部署需求”。物理视图描述运行软件的计算机、网络、硬件设施等情况，还包括如何将软件包部署（如果是嵌入式系统则是烧写）到这些硬件资源上，以

及它们运行时的配置情况。另外，物理架构还要考虑软件系统和包括硬件在内的整个 IT 系统之间是如何相互影响的，由于一部分运行时质量属性需要硬件或网络的支持，所以物理架构必须关注如何配置硬件和网络来配合软件系统的可靠性、可伸缩性、持续可用性、性能、安全性等方面的要求。相对于运行架构而言，物理架构重视目标程序的静态位置问题，而运行架构关注目标程序的动态执行情况。

物理架构还和数据的分布有关。如果说，系统=程序+数据+硬件，那么，物理架构就是要定义“程序”如何映射（安装、部署或烧写等）到“硬件”，以及“数据”如何在“硬件”上保存和传递。也就是说，物理架构必须考虑“功能的分布”和“数据的分布”这两个方面。

如果使用 UML 来描述架构的运行架构，则该视图可能包括部署图、组件图。

对分布式应用而言，物理架构常被称作部署架构。当然，如果考虑到嵌入式系统等的习惯，还是称为物理架构比较合理，毕竟此时目标模块是被“烧写”到硬件的，而不是一般理解上的“部署”。

一般而言，物理架构的设计应完成下列工作：

- 确定物理配置方案（可能是网络方案，也可能是单片机等的分布，或者二者兼有）
- 确定如何将目标程序映射到物理节点

## 16.7 注意满足所有约束性软件需求

这是一个真实的案例。

软件在航空电子系统中处于核心地位，它对军用飞机的战斗力、寿命周期、成本等均产生重要影响，而专用的嵌入式操作系统又是软件的基础。由于国家安全、军事竞争、量产规模等多方面的因素影响，航空领域的计算机远不如民用计算机领域那么“标准化”，很多硬件都是定制的，因此嵌入式操作系统必须将硬件层封装和屏蔽。由于负责该嵌入式操作系统的项目团队是第一次接触 VxWorks，架构设计师基于风险驱动的考虑，将 VxWorks 操作系统的剪裁作为重要技术风险，并特别派了精兵强将去攻克 VxWorks 的内存管理子系统……时光流转，到了总工听取项目预研情况的时候了，没想到总工一语惊醒梦中人：“你没想想内存一共才多大，你的操作系统就要占去 30%？”是的，还是总工看得清澈——把设计软件系统时的约束性要求看得清清楚楚。最后，制定的设计决策是：将 OS 的内存管理子系统全部剪裁掉，应用直接管理物理内存。

由上面的案例可以看出，约束性需求往往能够影响架构设计的方向，必须认真对待。

总之，约束性需求规定了开发软件系统时必须遵守的限制条件，忽视它们可能导致架构设计的失败。和功能性需求相比，约束并不强调行为，它既可能是设计开发时必须遵循的标准，也可能是硬件的限制，还可能是法律法规等社会因素的影响。例如，采用何种操作系统、采用何种开

发技术、需要和哪些遗留系统进行互操作等，可以视为技术性约束。而为了获得相关行业或组织的认可，或者大型企业集团出于长期整合规划的要求，软件的设计和开发可能还必须遵守相关的行业标准、企业标准等标准的约束。

## 16.8 PM Tool 实战：细化架构设计

如前所述，运用基于 5 多视图的架构设计方法时，各个架构视图的设计顺序没有严格规定。我们下面就从物理架构的设计开始，这对分布式应用往往比较合适。图 16-18 表达了 PM Tool 是如何采用 B/S 结构的，并确定了每个计算机节点上的主要软件单元的。

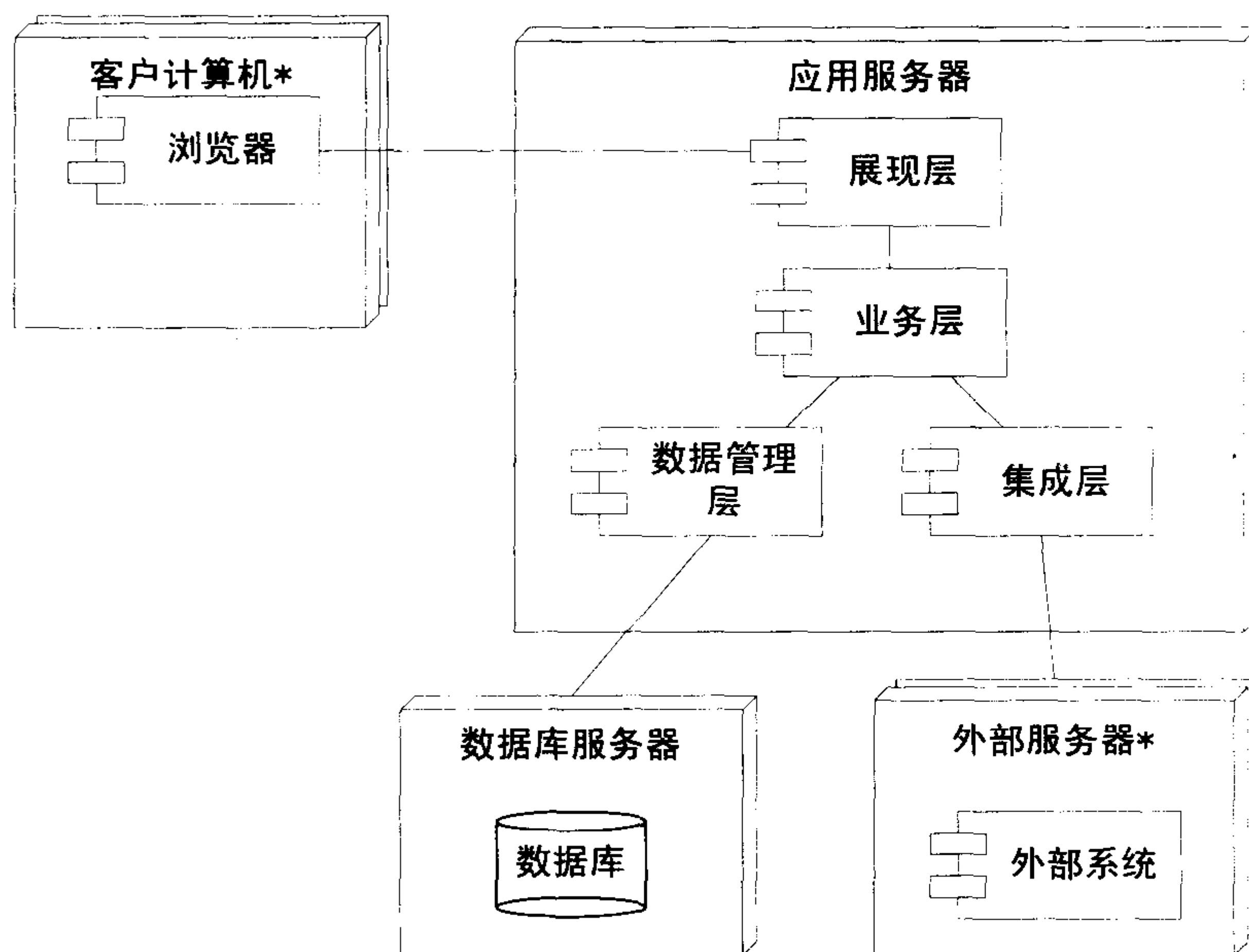


图 16-18 物理架构

对于逻辑架构而言，有非常多的设计工作要做。其中，识别架构的通用机制是非常重要的。我们可以通过研究每一层必须完成哪些职责、它们需要哪些协作，从而确定需要引入哪些通用机制和模式，表 16-3 列举了我们的一些设计决策。例如，我们决定对业务层采取“领域模型模式为主，事务脚本模式为辅”的设计，让不同模式分别来处理各自擅长的情况。又如，对于集成层而言，为了完成类似用 HR 系统获取 HR 资源列表等简单的互操作需求，只需采用最简单的“基于文件传输”的集成方式即可，而对“集成对 CVS 和 Subversion 的访问”这样更复杂的互操作，可以采用“基于远程调用”的集成方式。



表 16-3 确定引入哪些通用机制和模式

层次	通用机制与模式	如何考虑
展现层	MVC	多种展现方式的需要
业务层	领域模型模式（Domain Model）	模型的复杂性
	事务脚本模式（Transaction Script）	用于大批量数据查询、 便于优化
数据层	ORM	配合领域模型模式
	SQL Map	配合事务脚本模式
集成层	基于文件传输的集成	用于获取 HR 资源列表等简单的 互操作
	基于远程调用的集成	用于复杂的互操作

架构设计应该尽量采取一致的机制来解决相似的问题。但同时，也不应过于呆板，例如报表子系统采用事务脚本模式很合适，而采用领域模型模式会人为地把设计复杂化。

开发架构设计很重要的一点是，必须说明采用哪些具体技术。表 16-4 说明了 PM Tool 的开发架构决策。

表 16-4 决定具体开发技术

逻辑架构设计的决定		开发架构设计的决定
展现层	MVC	Struts
业务层	领域模型模式（Domain Model）	POJO
	事务脚本模式（Transaction Script）	POJO 结合 iBatis
数据层	ORM	Hibernate
	SQL Map	iBatis
集成层	基于文件传输的集成	Web 服务
	基于远程调用的集成	采用 Adapter 对 API 进行封装

另外，还应根据项目的具体情况（如团队对具体技术的熟悉程度）对重要的框架的用法进行说明。当然，框架本身的设计模型非常有用，但未必能说明我们要开发的软件系统将如何具体使用它。例如图 6-19 展示了 Struts 的运行机理，但我们需要结合待开发的软件系统来说明我们将如何利用它。

图 16-20 是项目开发所需要的，它运用了“分层 + 分区”的方法，明确了框架和待开发系统之间的关系。



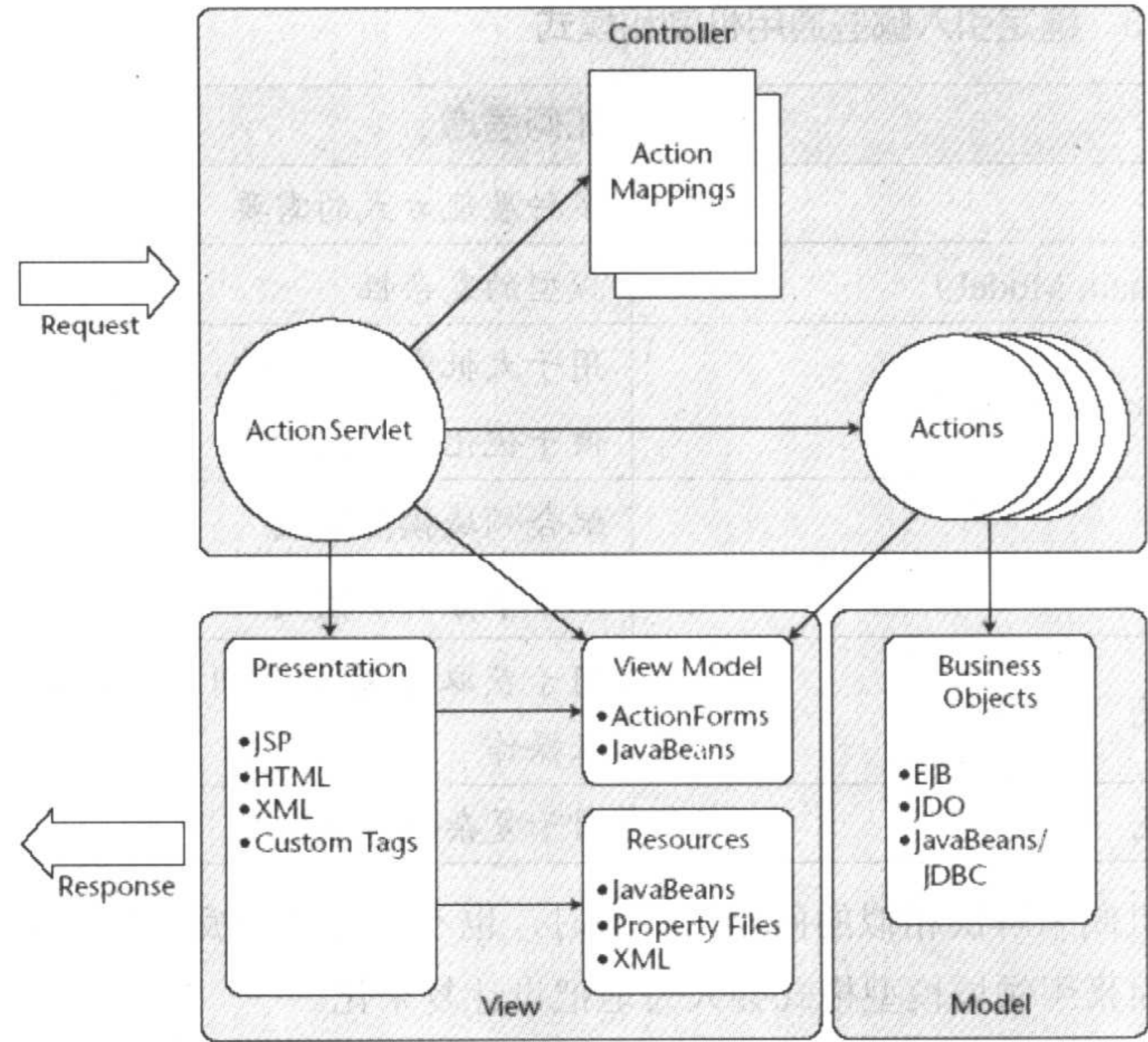


图 16-19 Struts 框架原理（图片来源：《Mastering JavaServer Faces》）

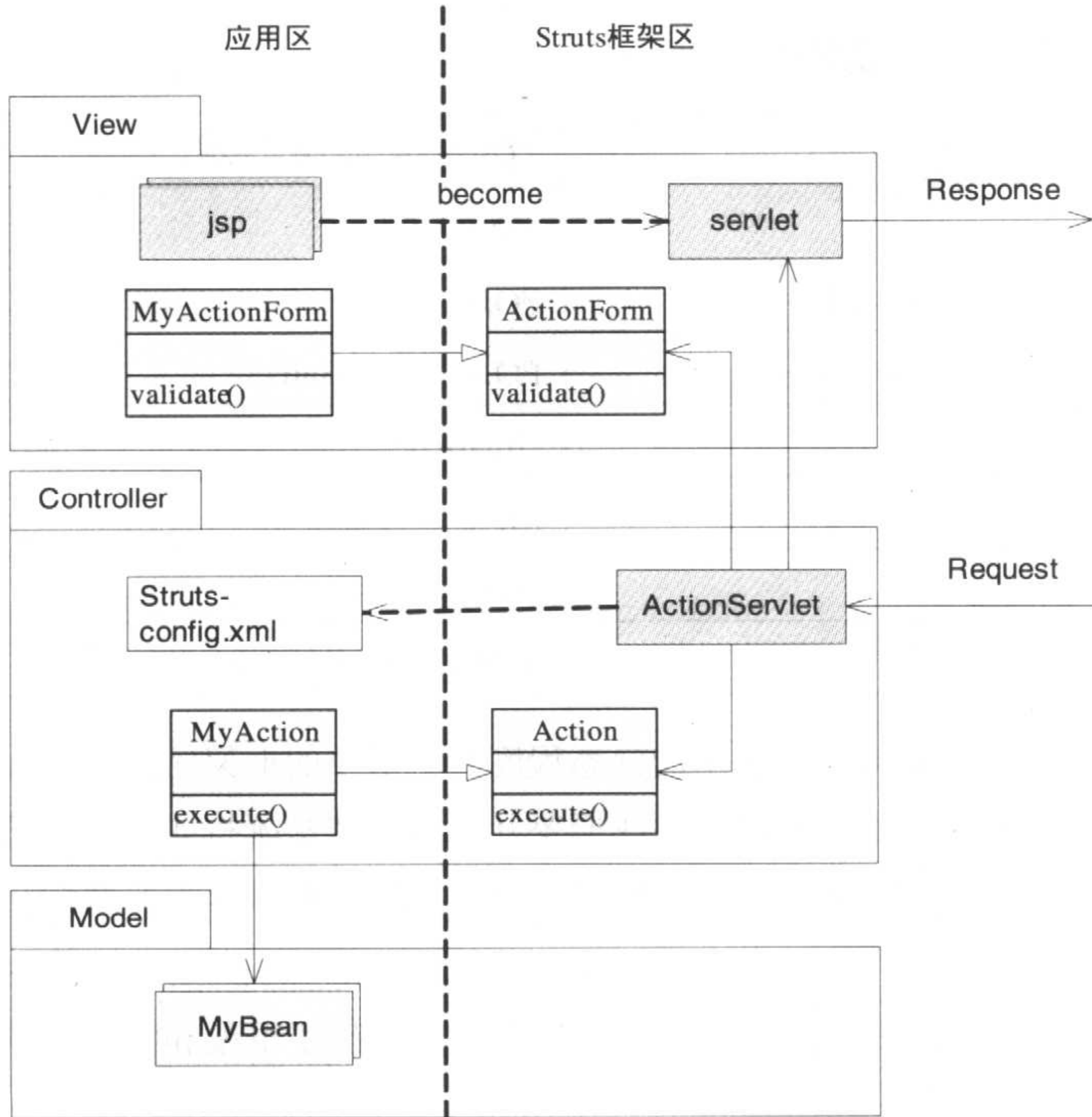


图 16-20 说明框架的用法

数据架构和运行架构的设计也非常关键，在此不再展开。

# 16.9 总结与强调

表 16-5 总结了运用架构设计的 5 视图进行架构细化设计的工作内容。

表 16-5 架构设计视图的设计任务

架构设计视图	设计任务
逻辑架构	细化功能单元 发现通用机制 细化领域模型 确定子系统接口和交互机制
开发架构	确定要开发或直接利用的程序包之间的依赖关系 确定采用的技术 确定采用的框架等
数据架构	持久化数据存储方案 数据传递、数据复制、数据同步等策略（可选）
运行架构	确定引入哪些进程与线程 确定主动对象、被动对象，以及控制流关系 处理进程线程的创建、销毁、通信机制、资源争用等 协议设计
物理架构	确定物理配置方案 确定如何将目标程序映射到物理节点





## 第 17 章 实现并验证软件架构

---

在耗费太多的钱和时间在我们推荐的体系结构上面之前，我们应该始终“问问电脑”对这个体系结构究竟怎么想。

——Rod Johnson, 《J2EE Development without EJB 中文版》

好的策略必须一再求证、测试、发现瑕疵漏洞，另想途径或方法来弥补策略不足，有时甚至得全盘放弃，重新策划。

——张明正, 《挡不住的趋势》

如果我们决定接受第一颗子弹，那么子弹到来得越早、越快就对我们越有利。

——Robert C. Martin, 《敏捷软件开发》

不值得验证的架构就不值得设计。

现代软件开发非常注重尽早降低风险。所谓风险，就是如果你忽略了它的存在，它就会主动找上门儿来的那种家伙。因此，我们必须主动防范风险。

架构设计是现代软件开发中最为关键的一环，架构设计得是否合理将直接影响到软件系统最终是否能够成功。毕竟，软件架构中包含了关于如何构建软件的一些最重要的设计决策，如系统分为哪些部分、各部分之间如何交互，等等；而采用这些设计决策，能否使最终开发出来的软件系统满足我们预期的运行期质量属性（如性能、可伸缩性、持续可用性、鲁棒性、安全性等）和开发期质量属性（如可扩展性、可重用性、可理解性）的要求，都是悬而未决的重大风险。

因此，在以架构为中心展开大规模的系统开发之前，切莫别忘记先“问问电脑”——也就是通过将软件架构设计方案尽快实现为一个小的原型，并通过对该原型的测试和评审，来评估软件架构是否合理。

## 17.1 基础知识

### 17.1.1 原型技术及分类

原型技术的思想是：对感兴趣的问题先试试看，而故意忽略其他方面的要求，从而使“试试看”的成本远远低于“正式干”的成本。

根据实验和评估的结果，我们可以做出很多有意义的判断：

- 我们担心的风险是否真的存在
- 通过开发原型是否找到了风险的解决办法
- 下一步，我们是让项目继续还是将项目取消
- 如果是悬而未决，我们是否要继续开发原型去探寻问题解决之道

具体而言，根据开发原型的目的是模拟系统运行时的概貌、还是纵深地验证一个具体的技术问题，可以将原型法分为两类：**水平原型**和**垂直原型**。

如图 17-1 所示，水平原型在一定程度上实现用户交互层的界面布局 and 界面流转逻辑。之所以说“或多或少”，是因为有高保真原型和低保真原型之分。低保真原型往往就是在白板上或文档中画出界面的草图。垂直原型一个形象的隐喻是“垂直切片”，它往往是涉及到不同的层，将为数不多的（甚至一个）功能真正地实现。

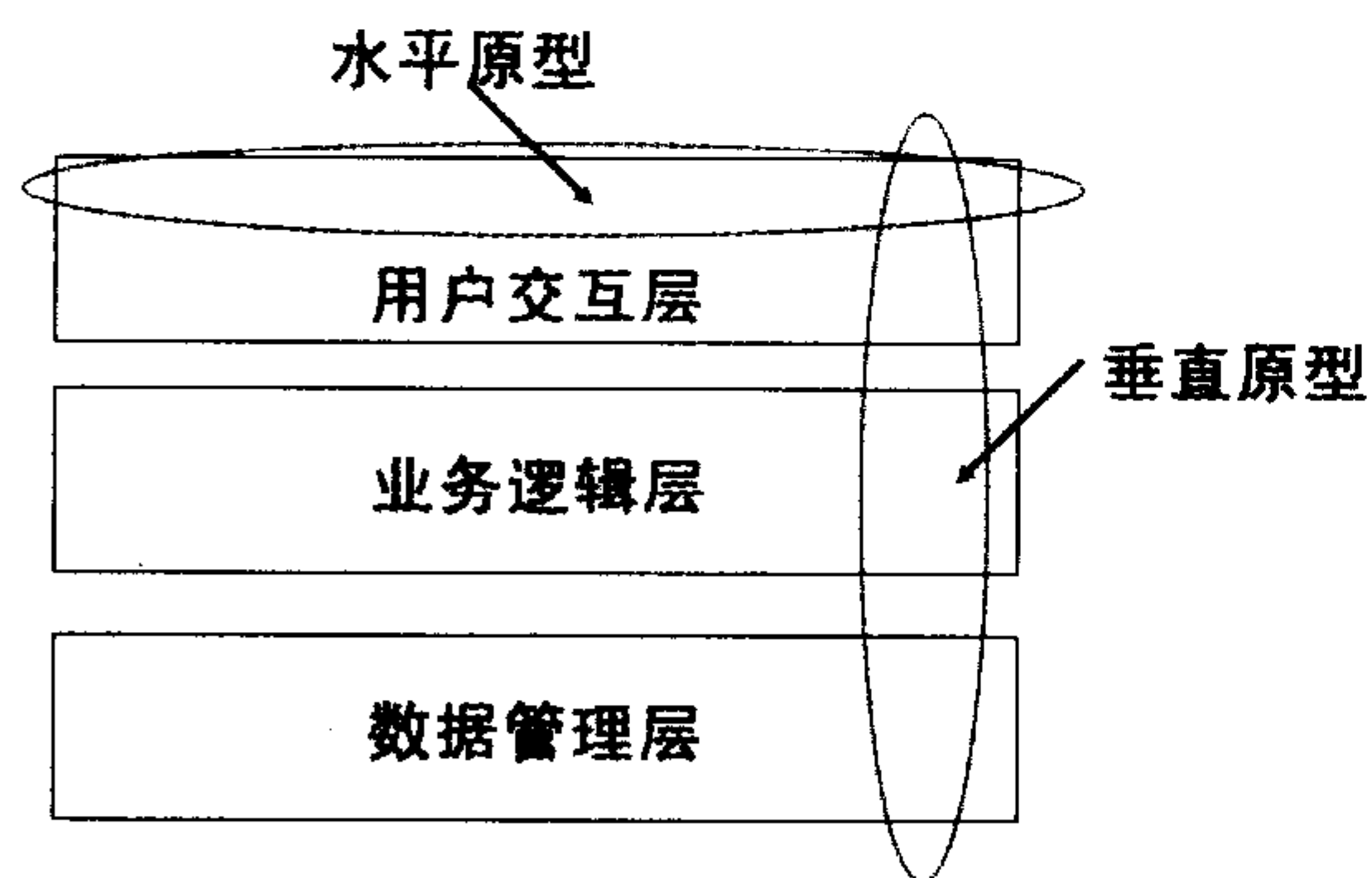


图 17-1 水平原型与垂直原型

另一方面，有些原型用过之后是注定要被抛弃的，而有些原型我们则希望把它们保留下来作为正式开发的基础。于是，根据所开发的原型是否被抛弃，我们又可以将原型法分为两类：**抛弃原型**和**演进原型**。

如表 17-1 所示，我们总结了原型法的两种相互独立的分类方式。



表 17-1 原型法的二维分类

	抛弃原型	演进原型
水平原型	水平抛弃原型	水平演进原型
垂直原型	垂直抛弃原型	垂直演进原型

水平抛弃原型很常用。有经验的开发人员（特别是系统分析员）都知道，大多数用户都不清楚他们想要什么样的系统，以及他们到底需要怎样的功能来辅助他们更好地完成工作任务；反倒是当切实地看到类似的软件系统时，通过他们的反馈和评价获取的信息中蕴藏着更多真实的需求——哪怕是他们看到了极其不喜欢的系统原型，他们也会表达出类似“我更喜欢如何”这样有价值的信息。因此，可以开发水平抛弃原型，模拟目标系统的界面布局和界面流转情况；通过观察用户模拟使用该原型系统的情况、听取用户的评价、和用户展开讨论等手段，来启发和捕获用户的真正需求。

正如前文所述，水平抛弃原型非常强调“通过故意忽略其他方面的要求来降低原型开发的成本”。的确如此，大多数用于需求启发的水平抛弃原型只注重模拟待开发系统的界面布局和界面流转情况，真正的业务逻辑都未实现，也没有任何数据库的连接生效。从而，我们可以采用如下方式快速地开发水平抛弃原型。

- 可视化设计工具：FrontPage、DreamWeaver 等；
- RAD 开发环境：VB、Delphi、JDeveloper 等；
- 脚本语言：PHP、JSP、Python、甚至 Html 等；
- 白纸加画笔：或者用 Photoshop、Visio 或 Word 甚至 Excel 来代替。

图 17-2 是一个人事管理系统水平原型的例子，其中展示了 2 个页面（员工列表页面和员工详细信息页面）的界面布局，还展示了 1 个界面流转（从员工列表页面到员工详细信息页面的转换）是如何被触发的。

水平演进原型似乎比水平抛弃原型少见得多，但这并不意外着它并不存在。很典型地，它往往和特定的 RAD 开发环境或特定的 Framework 的支持并存：

- 当采用 VB 或 Delphi 开发时完全可以这样做。（RAD 模式的开发现在好像冷落下来了，但它并没有消失，而是暂时潜伏下来，等待模型驱动开发的真正成熟。）

笔者认为，引领 Web 界面开发的面向对象风潮的 JSF 技术、Tapestry 框架等的出现，又为“高调应用”水平演进原型提供了机会——它们将为不懂得编程的美工通过“所见即所得”方式开发的界面原型“演进”成软件系统的一部分提供了基础。

当然，典型并不代表绝对，水平演进原型可以有别的用法。例如，经过多次水平抛弃原型的探索和逼近，所模拟开发的原型和最终系统的原型已经非常接近了，如果所采用的开发语言是一致的，那么可以把后期的原型作为演进原型使用。



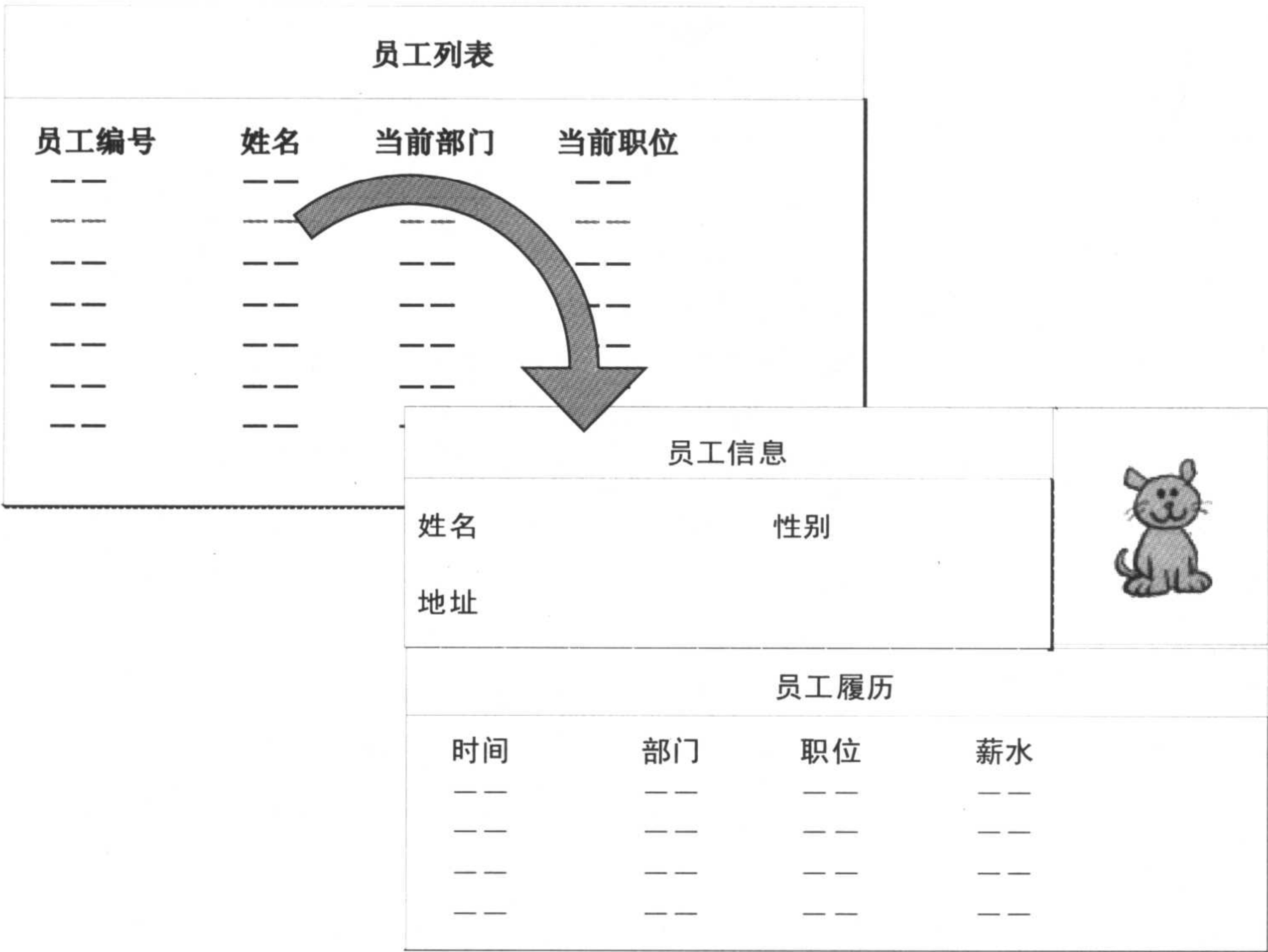


图 17-2 人事管理系统水平原型示意图

垂直抛弃原型常用来进行技术验证。在很多项目中，常会有一些项目组并不熟悉的技术点，如果这些技术点对项目的成功与否影响重大，那么你应该通过开发垂直抛弃原型的方法，先演练演练它是否能帮助你解决预期的问题。

举个垂直抛弃原型的例子。笔者曾负责过一个 J2SE 的项目，其间，我们决定基于 J2SE 为自己的应用开发一个应用框架（相信不少读者听到过类似“J2SE 没有提供应用框架（Application Framework）”的批评吧！）。正如你所知，用户可以通过多种手段发现指令给软件系统（就是 MVC 所指的多种 Controller）：

- 菜单中的菜单项
- 工具栏中的按钮
- 快捷键

每种手段都要通过编程的方式和处理逻辑挂接起来，这对大型应用程序而言很不方便。因此我们决定在应用框架中提供“数据驱动”的方法来进行控制方式和处理逻辑的挂接，这需要使用反射（Reflection）技术来支持处理逻辑类的实例化。为此，我们开发了一个小的垂直抛弃原型，来验证反射机制是否能够帮助我们达到预期的目标。这个原型使我们实现了尽早降低风险的目标。



垂直演进原型大家都很熟悉。其实，它和迭代式软件开发所暗示的“多次交付，增量交付”的思想完全一致。

垂直演进原型方法强调逐步提供用户所要求的功能。在实践中，我们应当注重每个此次将提交的功能必须是可用的并具有或接近产品发布级的产品质量，而不是有些实践者所认为的无需达到发布级质量。垂直演进原型方法使我想起了句流行语，这就是比“早发布，常发布”更激进的“永远只是 Beta 版”；其实，永远 Beta 版更多地强调的是整个产品的功能并未最终冻结，传统意义上的“Beta 版”的用户级测试的含义倒是位于其次了。

最后，介绍每种原型法的“外号儿”：

- 水平原型。又称行为原型；
- 垂直原型。又称结构原型；
- 抛弃原型。又称探索原型；
- 演进原型。似乎没有“外号儿”，不过它和增量开发的思想相同。

### 17.1.2 验证架构的两种方法

具体而言，验证架构有两种方法——**原型法**和**框架法**，每种方法所适用的情况有所不同。

对于项目型开发，常采用“原型法”。其实准确而言是“垂直演进原型”：为了真实地验证架构的表现，必须将选定的功能特性完整地实现；另一方面，这个原型不是验证单个技术的运用是否可行的垂直抛弃原型，而是要对一组架构设计决策“对系统要求的非功能需求的满足程度”进行验证，所以这个垂直原型应该是演进型的，将直接作为后面分头开发的基础。有些读者可能会对 RUP（统一软件过程，Rational Unified Process）中提到的“可执行架构”感到困惑，其实它就是这里所讲的垂直演进原型。

对于产品型开发，采用“框架法”有更多优点。所谓“架构验证的框架法”就是将架构设计方案用框架的形式实现，并在此基础上进行评估验证。引入框架之后，整个“应用空间”的分割多了一个维度——框架实现与具体应用无关的通用机制和通用组件，这利于支持产品型开发的生命周期长、应用版本多等特点。当然，由于框架本身并不提供任何具体的应用功能，所以在把架构设计思想框架化之后，应在框架基础上实现部分应用的功能——即实现一个小的垂直原型，从而进行实际的非功能测试和开发期质量属性评价。

## 17.2 实现并验证软件架构的具体做法

架构原型对功能性需求的实现非常有限，那么我们“架构验证”要验证什么？答案是要验证架构对质量属性需求的支持程度：

- 在“放手”实现功能需求之前，我们必须通过实际的测试，来验证架构对运行期质量

属性的支持程度是否达到了要求，这是因为这些质量属性并不单纯和某个功能有关、而是系统架构的大局规划决定的。例如，如果通过测试，发现性能或可伸缩性达不到要求，就必须尽快调整架构设计；

- 架构设计方案还对开发和维护工作造成深远影响，所以必须尽早验证架构在开发期质量属性方面的表现，具体方法是让参与架构原型开发的人员进行评估。例如，架构设计方案的可理解性如何，基于此架构开发出的应用（当前是原型）的可测试性、可重用性如何等等。

图 17-3 归纳了验证架构的具体步骤。首先，必须将架构设计方案付诸实现，得到的架构原型可以是纯粹的垂直演进原型，也可以是基于架构框架的原型。之后，分头进行运行期和开发期质量属性的测试或评审，分别得到运行期质量的测试结果和开发期质量的评审结果。最后，判定架构设计是否合乎要求，如果不合格，应决定下一步需要对架构的哪些方面进行重新设计。

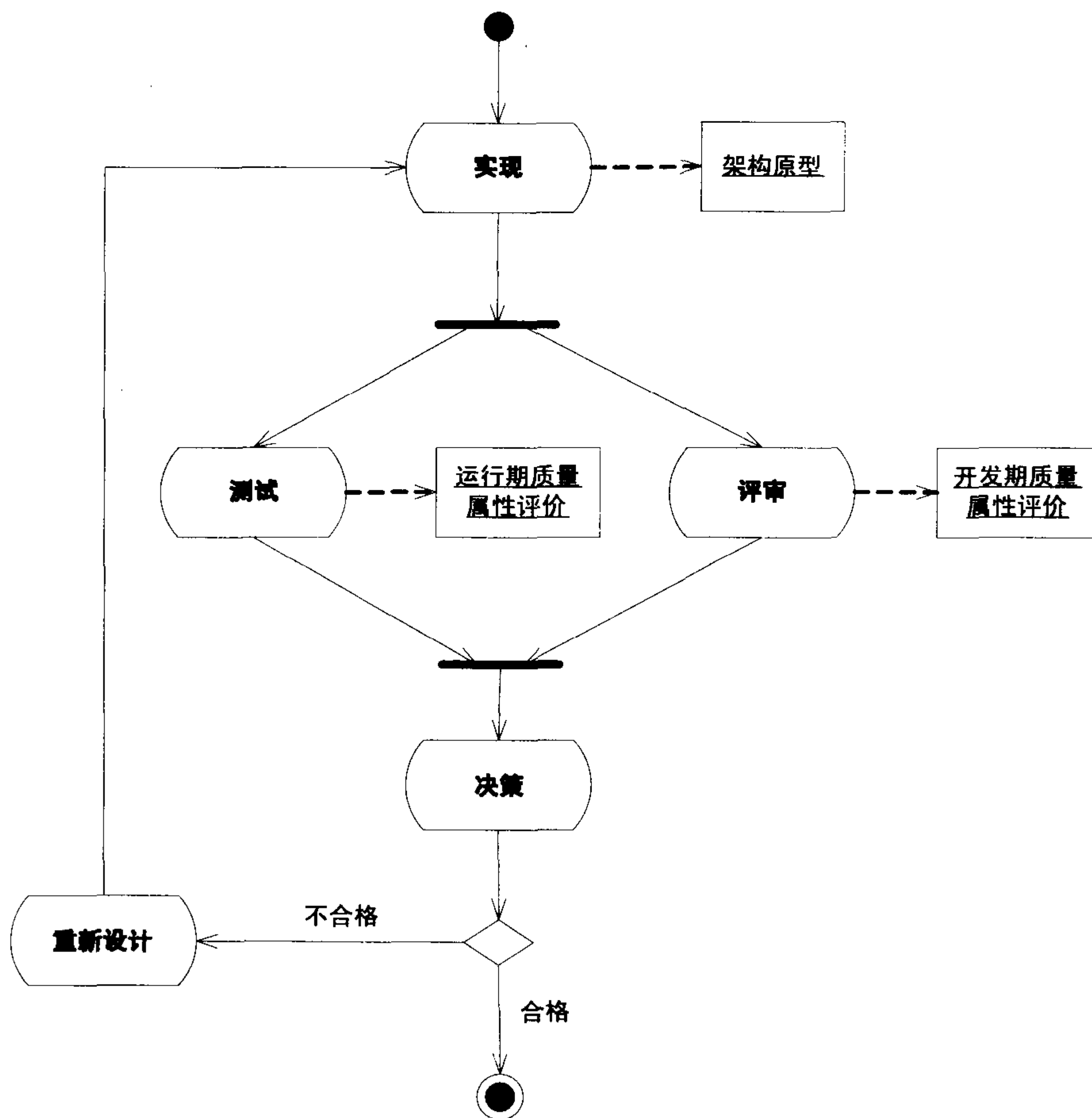


图 17-3 验证架构的具体步骤

实现架构原型时，代码要达到产品级质量，因为架构原型不是抛弃原型，而是作为演进原型，并且要成为下一步开发的基础。

当然，架构原型所实现的有限的功能需求应经过细心挑选，它们应该是能够“触发”主要的设计机制参与执行的、或有较高技术风险的、或最影响用户满意度的一些功能。一句话，这些功能要么是用户“最关心的”，要么是架构师“最担心的”。

我们必须让架构“跑”起来，从而对它进行真实的测试，测试的重点是质量属性的测试，而不是功能测试。为此，我们必须“营造”出想要的场景，可能要借助专门的工具或进行插桩测试等，在此不再展开。

最终，通过验证的架构设计方案就可以作为大规模开发的基础了，这时投入大量资源是合适的，因为重大的技术风险已在架构设计中得到了解决和验证。

## 17.3 总结与强调

---

回忆软件架构到底为谁而设计的问题……

既然软件架构师必须内外兼顾、各层并重（如图 4-3 所示），而软件架构又是如此重要——它包含了关于如何构建软件的一些最重要的设计决策，所以就有必要进行实实在在的验证：让实际测试结果告诉我们它的运行期表现，让开发人员告诉我们它的开发期表现。



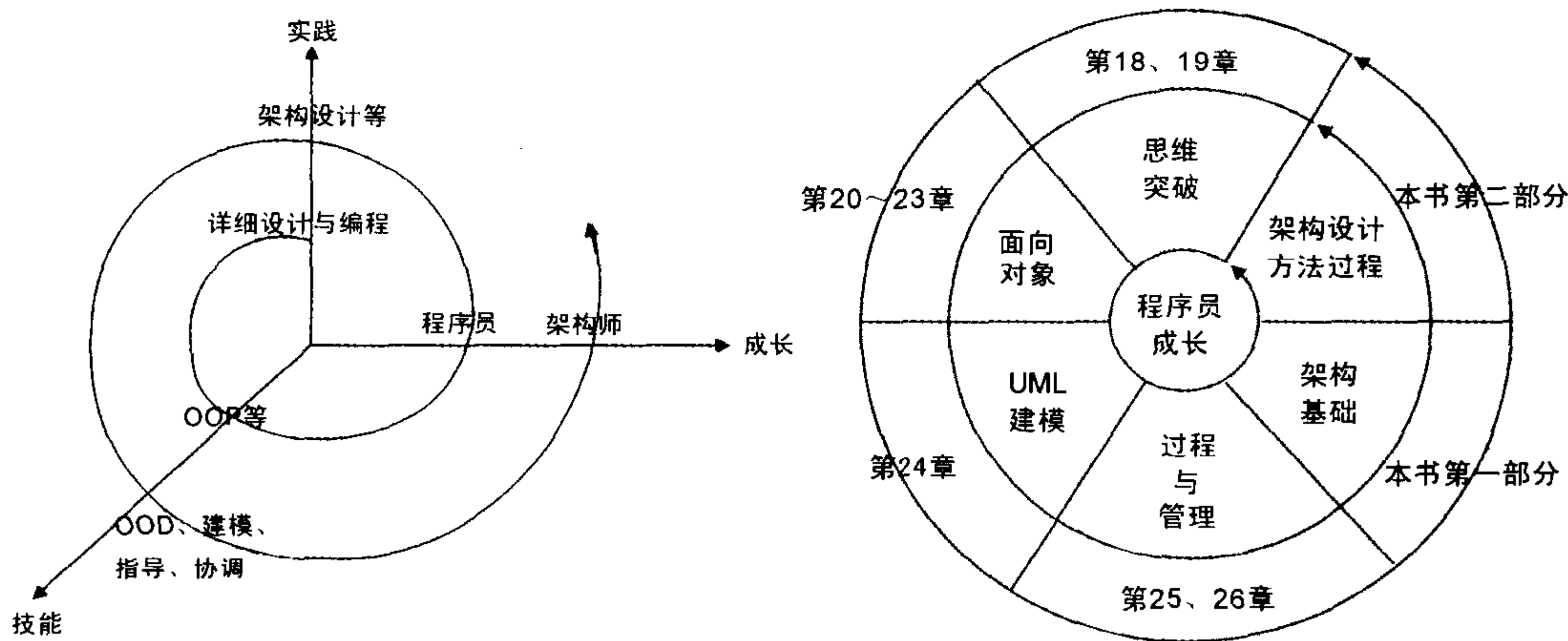


# 第三部分

## 程序员成长篇

如果说，本书“第二部分 软件架构设计方法与过程篇”是以“自顶向下”方式讲解如何设计软件架构的话，那么即将开始的第三部分就是“自底向上”地说明从程序员到架构师的成长过程中，哪些关键思维必须突破，哪些重要技能必须掌握。

本部分共包含 9 章，内容围绕思维突破、面向对象、UML 建模、过程与管理这 4 个程序员成长中必须关注的方面。





# 第 18 章 MIME 编码类案例：

## 从面向过程到面向对象

---

使用传统的过程化程序设计所创建出来的依赖关系结构，策略是依赖于细节的。这是糟糕的，因为这样会使策略受到细节改变的影响。

——Robert C. Martin, 《敏捷软件开发：原则、模式与实践》

习惯很重要。《高效能人士的七个习惯》整本书都在讲习惯。心理学的研究也表明，人在面临巨大压力的时候，以及在一点儿压力也没有的时候，其行为方式往往是由习惯决定的。

在软件开发中习惯也很重要。比如，一旦项目压力大的时候，开发活动就不再那么“讲究”了，也不再强调面向对象了，面向过程就面向过程吧，按时完成项目要紧。分析起来，这其中的主要原因并不是面向过程比面向对象简单，而是由于人们在压力巨大的时候会不自主地选择自己所习惯的做事方式。换句话说，那些已经习惯面向对象思维的开发人员，是不会因为项目压力大而“滑向”面向过程方法的。

我们应该真正建立起面向对象的思维习惯，因为它早已超越了面向过程方法而成为软件开发方法的主流。作为希望成为软件架构师的程序员，突破面向过程思维、建立面向对象思维是非常基础的一步。这正是本章的主题。

至于讲述方式，本章将采取“现身说法”，比较全面地讲述一个案例：不仅包含项目背景、项目目标、最终设计方案的介绍，还包含了从面向过程设计思路一步步往面向对象设计思路转变的思考过程——毕竟，直接讲述设计方案并非最好的学习方法，讲述整个设计的思维过程才是。

### 18.1 设计目标

---

我们计划开发一套 MIME 编码和解码的类，适用于下列多种应用场合：

- E-mail 客户端程序
- 乱码察看程序
- 图片等二进制对象存入 XML 文件

设计目标如下：

- 可重用
- 易使用
- 易扩充

## 18.2 MIME 编码基础知识

MIME 是一种 Internet 协议，全称为“Multipurpose Internet Mail Extensions”，中文名称为“多用途互联网邮件扩展”。其实，它的应用并不局限于收发 Internet 邮件——它已经成为 Internet 上传输多媒体信息的基本协议之一。本章仅关心 MIME 的编码算法。

MIME 编码的原理就是把 8 bit 的内容转换成 7 bit 的形式以能正确传输，在接收方收到之后，再将其还原成 8 bit 的内容。对邮件进行编码最初的原因是因为 Internet 上的很多网关不能正确传输 8 bit 内码的字符，比如汉字等。MIME 编码共有 Base64、Quoted-printable、7bit、8bit 和 Binary 等几种。

Base64 算法将输入的字符串或一段数据编码成只含有{'A'-'Z', 'a'-'z', '0'-'9', '+', '/'}这 64 个字符的串，'='用于填充。其编码的方法是，将输入数据流每次取 6 bit，用此 6 bit 的值(0-63)作为索引去查表，输出相应字符。这样，每 3 个字节将编码为 4 个字符( $3 \times 8 \rightarrow 4 \times 6$ )；不满 4 个字符的以 '=' 填充。

Quoted-printable 算法根据输入的字符串或字节范围进行编码，若是不需编码的字符，直接输出；若需要编码，则先输出 '=', 后面跟着以 2 个字符表示的十六进制字节值。

## 18.3 MIME 编码类的设计过程

本节将分为 3 个小节。注意它们不是并列的 3 种设计方案，而是达到趋于合理的设计的思考过程。

### 18.3.1 面向过程的设计方案

首先跳进我脑子的想法就是设计成 Utility。

Utility 被称为实用类，虽说也冠以“类”之美名，但它却是仅提供方法的类，说到底还是面向过程的思想。很多有过面向过程开发经验的人，以及没有面向对象开发经验的新手，他们解决问题的最初思路往往都是基于面向过程方法的。对此，GOF 在《设计模式》中怎么说来着？

“有经验的面向对象设计者的确能做出良好的设计，而新手则面对众多选择无从下手，总是求助



于以前使用过的非面向对象技术。”看来此话不假。

将这个 Utility 命名为 UMime 吧！那么它包含哪些方法呢？差不多像这样：

```
bool UMime::Encode(unsigned char * outTargetBuf,
    int & outTargetBufLen,
    const unsigned char * const inSourceBuf,
    int inSourceBufLen);
```

```
bool UMime::Decode(unsigned char * outTargetBuf,
    int & outTargetBufLen,
    const unsigned char * const inSourceBuf,
    int inSourceBufLen);
```

不行，为了满足“易使用”的要求，应该支持 CString 类型的 buffer 吧！再增加如下 2 个方法：

```
bool UMime::Encode(CString & outTargetStr,
    CString & inSourceStr);
```

```
bool UMime::Decode(CString & outTargetStr,
    CString & inSourceStr);
```

这样一来，UMime 实用类就一共包含 4 个方法了。

好像还不错？

高兴得太早了。因为将来应用中很可能出现 CString 和 unsigned char \* 协同工作的情形。比如应用从 XML 文件中读出一个字符串放到一个 CString 型变量中，而这个字符串是一个 Bmp 图片的 MIME 编码，它解码过后自然应放到 unsigned char \* 的 buffer 中。所以我们还要增加下面 4 个方法才能免去用户的类型转换之苦：

```
bool UMime::Encode(CString & outTargetStr,
    const unsigned char * const inSourceBuf,
    int inSourceBufLen);
```

```
bool UMime::Decode(CString & outTargetStr,
    const unsigned char * const inSourceBuf,
    int inSourceBufLen);
```

```
bool UMime::Encode(unsigned char * outTargetBuf,
    int & outTargetBufLen,
    CString & inSourceStr);
```

```
bool UMime::Decode(unsigned char * outTargetBuf,
    int & outTargetBufLen,
    CString & inSourceStr);
```

啊哈！这么 8 个极为相似的方法搅在一起，好像一团麻呀！可重用性似乎满足了，但易使用性和易扩展性完全谈不上。

### 18.3.2 转向面向对象设计

面向过程思维所产生的设计方案并不令人满意。看来，需要寻求更合理的设计。

第 2 种方案浮现在脑海中：

- 既然整个算法就是将一个 Buffer 转换成另一个 Buffer，写成一个 String Class 是非常自然的设计。
- 用 Class 的成员变量保存 Target Buffer 及其长度（因为 Buffer 中可能有'\0'，所以需要“长度”这个属性）。
- 另外，应提供 GetBuf( )和 GetBufLen( )等方法来支持对 Target Buffer 的查询。
- 最好直接从构造函数传递 Source Buffer 的信息，这样可以使接口显得更加简洁。

该类大概像这样：

```
class CMimeString
{
public:
    enum PROCESSTYPE
    {
        ENCODING = 0,
        DECODING = 1
    };
public:
    CMimeString(PROCESSTYPE inType,
        const unsigned char * const inBuf,
        int inBufLen);
    CMimeString(PROCESSTYPE inType, CString & inStr);
    virtual ~CMimeString( );
    unsigned char * GetBuf( void );
    int GetBufLen( void );
    operator LPCTSTR( ) const;
};
```

哈，似乎很美妙。

- Source Buffer 仍然支持 unsigned char \*和 CString 这两种类型，而 Target Buffer 由 CMimeString 本身来管理不必用户操心了。
- 但具体应用不是对二进制对象进行编码时，可以不用 foo( s.GetBuf( ) )而直接用 foo( s )，因为 operator LPCTSTR( ) const;自动负责类型转换。
- 直接从构造函数传递 Source Buffer 的信息，使得接口更为精简。

当具体使用 CMimeString 时，大概像这样：

```
CString buf("sadsdfsdf");
CMimeString mime(CMimeString::ENCODING, buf);
MessageBox( s );
```

看来易使用性不错，下面要着重解决易扩展性了。CMimeString 的实现部分会像这样：

```
class CMimeString
{
.....
protected:
    unsigned char * mBuf;
    int mBufLen;
    virtual void Encode( unsigned char * inBuf, int inBufLen );
    virtual void Decode( unsigned char * inBuf, int inBufLen );
};
```

其中的两个虚函数是专门为易扩展性准备的，要实现新的 MIME 编码算法，只需要从 CMimeString 继承一个子类。例如，当我们要实现 Base64 算法时，只需从 CMimeString 继承一个名为 CBase64String 的子类即可（类图如图 18-1 所示）：

```
class CBase64String : public CMimeString
{
protected:
    virtual void Encode( unsigned char * inBuf, int inBufLen );
    virtual void Decode( unsigned char * inBuf, int inBufLen );
};
```

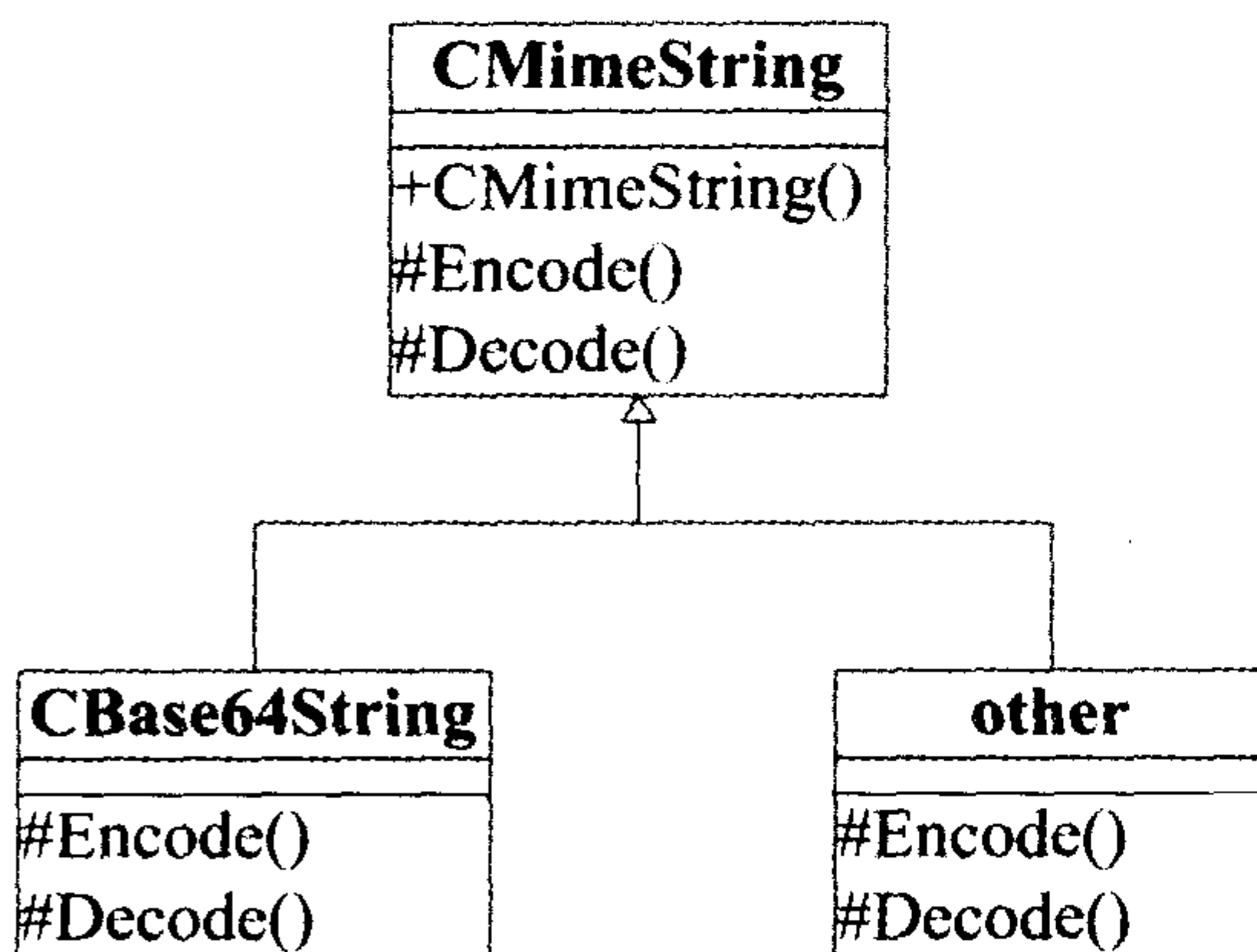


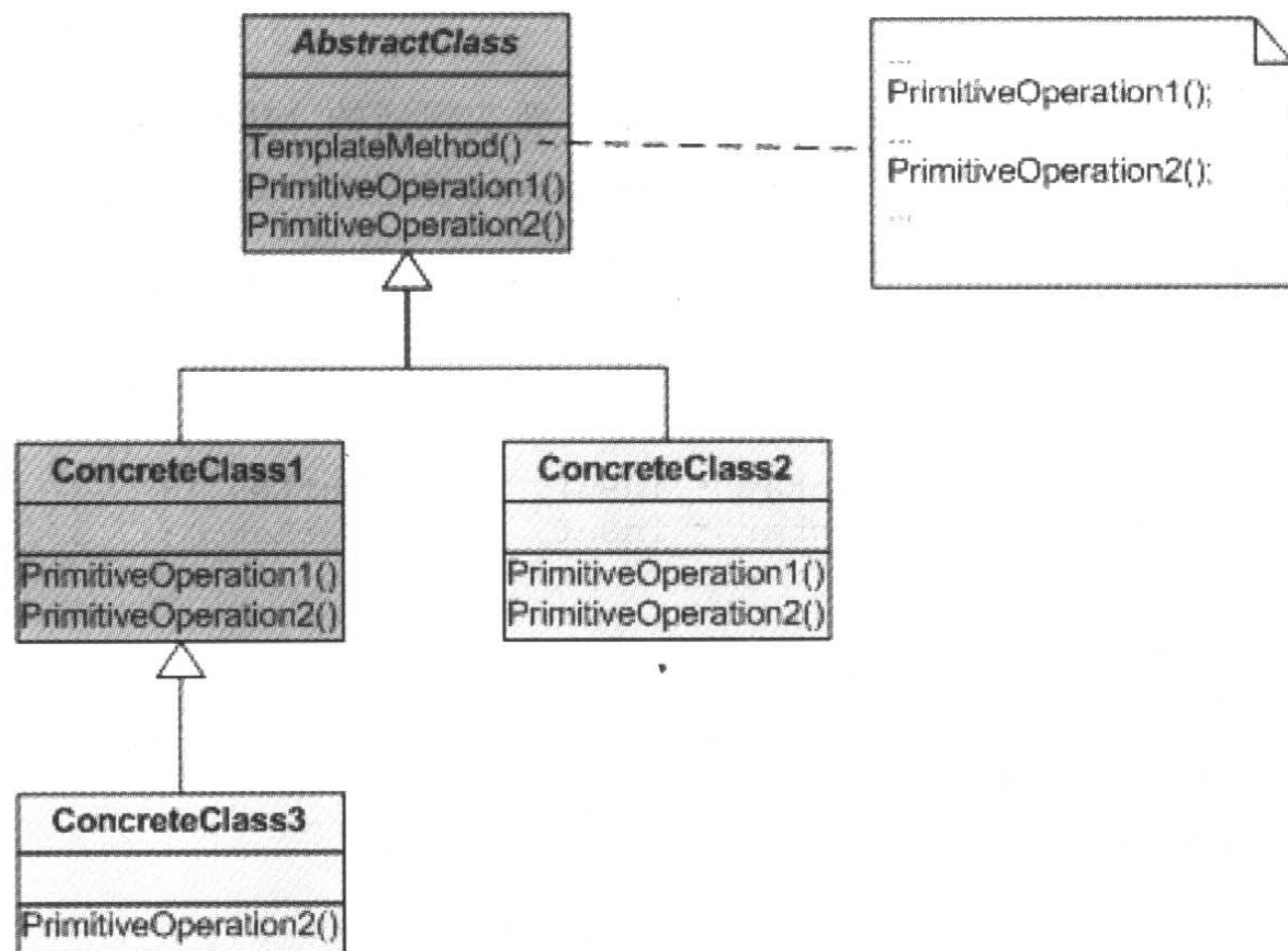
图 18-1 采用 Template Method 模式的设计

## 模板方法（Template Method）模式

关键字：算法骨架

支持变化：子类可以只改变算法的特定步骤，而不改变和继续使用算法骨架。编写子类的工作量很小，只需 Override 相应的 Virtual 函数。很多框架（Framework）都借助此模式实现了反向控制。

结构：



哈哈，我似乎看到 Template Method 模式向我招手了，让我盘算盘算。嗯……Encode( )和 Decode( )这两个虚函数是在哪里被调用的呢？在基类的构造函数中：

```

CMimeString::CMimeString(WHICHTYPE inType, CString & inStr)
{
    mBuf = 0;
    mBufLen = 0;
    if( inType == ENCODING )
    {
        Encode((unsigned char *) (inStr.operator LPCTSTR( )),
inStr.GetLength( ));
    }
    else if( inType == DECODING )
    {
        Decode((unsigned char *) (inStr.operator LPCTSTR( )),
inStr.GetLength( ));
    }
}

```



看上去是很不错的 **Template Method** 模式的运用，但是这根本就不会产生我们想要的结果——因为“在构造函数中调用虚函数”并无多态特性！

```
CBase64String::CBase64String(PROCESSTYPE inType, CString inStr)
{
    OnlyInitSelf( );
}
```

之后

```
CString buf("sadsdfsdf");
CBase64String base64(CMimeString::ENCODING, buf);
MessageBox( base64 );
```

是不对的，仍然是基类的 `CMimeString::Encode()` 被调用了，而且 `OnlyInitSelf()` 在 `Encode()` 被调用之后才被调到。

看来，这个设计还是不行。

### 18.3.3 面向对象设计方案的确定

是不是有些懊恼？别急。分析问题背后的问题：我们想用 **Template Method** 模式倒是没错，但是让构造函数扮演 **Template Method** 的角色是错误的，它先天（C++本身决定的）就不是这块料。

现在，摆在面前的至少有两种方法：

- 第 1 种方法是，坚持使用 **Template Method** 模式，但要增加一个方法扮演 **Template Method** 角色。这样一来，我们使用 `CMimeString` 时就不如“直接从构造函数传递参数”方便；
- 第 2 种方法是，坚持直接从构造函数传递参数，放弃 **Template Method** 模式，改用其他模式完成“改变算法”的职责。

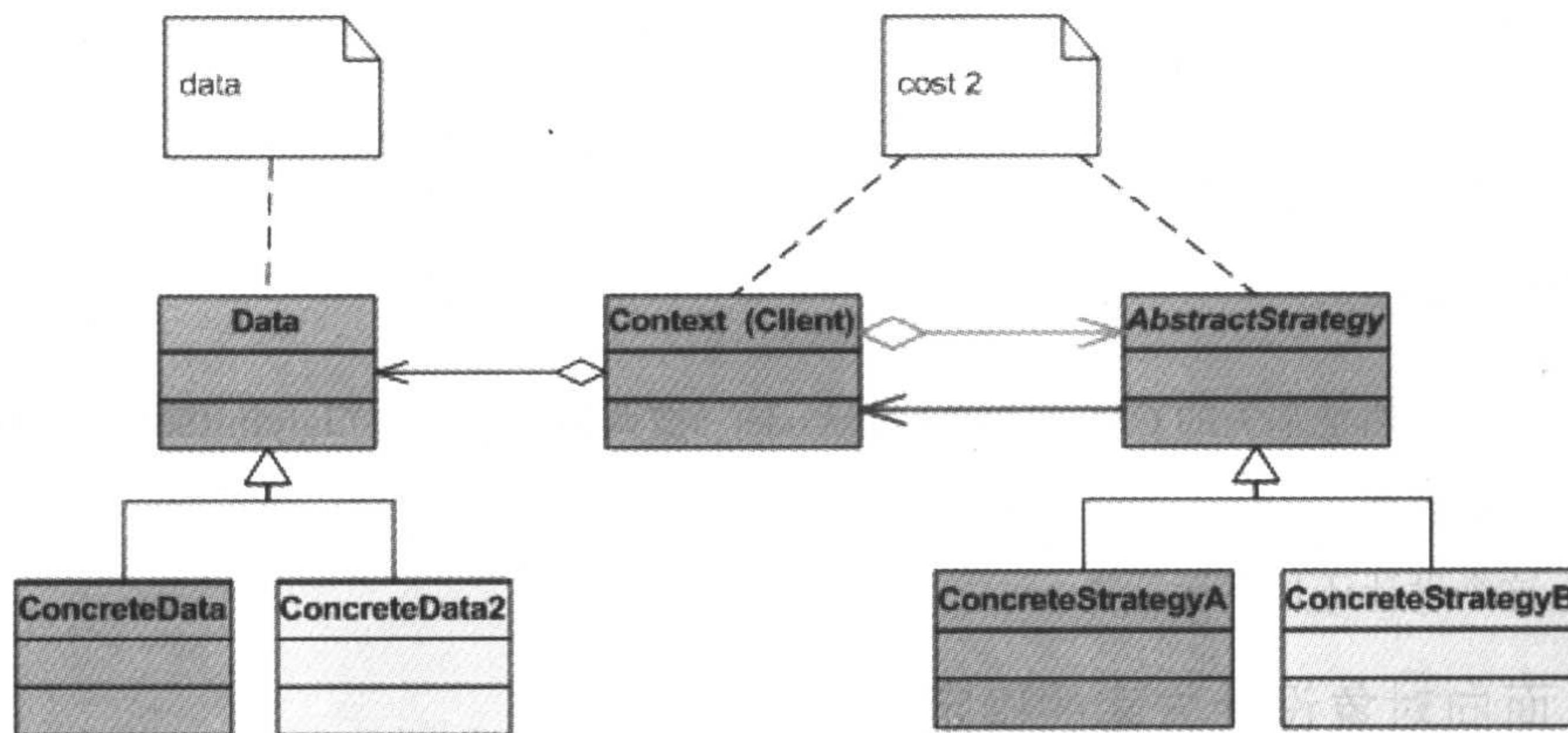
我决定采用第 2 种方法。

## 策略 ( Strategy ) 模式

关键字：算法族

支持变化：它使得算法可以独立于使用它的客户而变化，多个算法之间也可以相互替换。

结构：



除了 Template Method 模式以为，Strategy 模式也可以履行“改变算法”的职责，我们就用 Strategy 模式代替 Template Method 模式继续完成 CMimeString 的设计（类图如图 18-2 所示）：

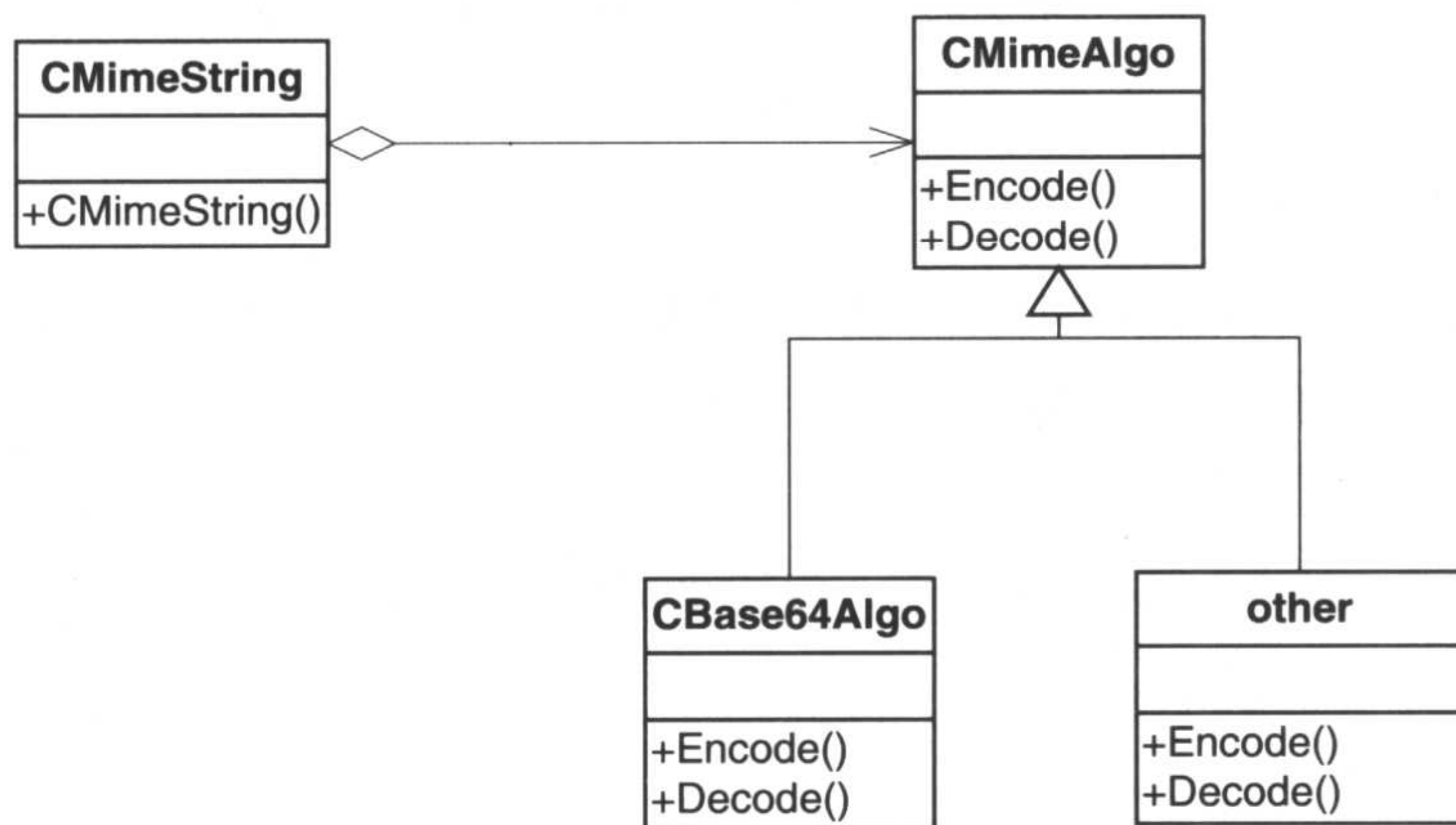


图 18-2 采用 Strategy 模式的设计

新的 CMimeString 的类声明如下：

```

class CMimeString
{
public:
    enum PROCESSTYPE
    {
        ENCODING = 0,
    }
};

```

```

    DECODING = 1
};
enum ENCODETYPE
{
    WYMIME = 0,
    BASE64 = 1
};
public:
    CMimeString(PROCESSTYPE inType,
        ENCODETYPE inAlgoType,
        CString & inStr);
    CMimeString(PROCESSTYPE inType,
        ENCODETYPE inAlgoType,
        unsigned char * inBuf,
        int inBufLen);
    virtual ~CMimeString( );
    int GetBufLen(void);
    unsigned char * GetBuf(void);
    operator LPCTSTR( ) const;
};

```

**CMimeAlgo** 的类声明如下:

```

class CMimeAlgo
{
public:
    CMimeAlgo( );
    ~CMimeAlgo( );
    virtual void Encode( unsigned char ** outBuf,
        int & outBufLen,
        unsigned char * inSrcBuf,
        int inSrcLen );
    virtual void Decode( unsigned char ** outBuf,
        int & outBufLen,
        unsigned char * inSrcBuf, int inSrcLen );
};

```

子类 **CBase64Algo** 的类声明如下:

```

class CBase64Algo : public CMimeAlgo
{
public:
    CBase64Algo( );
    ~CBase64Algo( );
    virtual void Encode( unsigned char ** outBuf,
        int & outBufLen,
        unsigned char * inSrcBuf,
        int inSrcLen );
    virtual void Decode( unsigned char ** outBuf,
        int & outBufLen,
        unsigned char * inSrcBuf,
        int inSrcLen );
};

```

具体使用 **Base64** 算法时会像这样:

```
CString buf("sdfsdsdfsdfsdf");  
CMimeString base64( CMimeString::ENCODING, CMimeString::BASE64, buf );  
MessageBox(base64);
```

哈哈，对此设计基本满意。

### 18.3.4 Template Method 和 Strategy 模式的对比

案例讲完了，随便对 Template Method 和 Strategy 这两个模式进行对比：

- 相同点。都是行为型模式，目的都是为了方便地改变算法；
- 不同点。实现方式不同，前者使用继承，称为类模式；后者使用委托，称为对象模式。

《设计模式》一书在讲到 Template method 模式和 Strategy 模式的关系时说：“模板方法使用继承来改变算法的一部分。Strategy 使用委托来改变整个算法。”

其中提到了“算法的一部分”和“整个算法”的区别。笔者认为，“整个算法”是“算法的一部分”的特例（就像数学中全集是集合的特例）——其实在实践中模板方法中又调用模板方法的例子太多了（例如很多开源框架就是这样）——因此是“算法的一部分”还是“整个算法”算不上这两个模式最根本的区别。倒是“继承”和“委托”的区别，即“类模式”和“对象模式”的区别，更本质地将这两个模式区分开来。

顺便说明，《设计模式》一书中非常强调对象模式和类模式的区别，本章就提供了一个很极端的例子——用对象模式可行而用类模式不可行。



## 第 19 章 突破 OOP 思维：继承在 OOD 中的应用

---

老僧三十年前未参禅时，见山是山，见水是水。及至后来亲见知识，有个入处，见山不是山，见水不是水。而今得个休歇处，依前见山只是山，见水只是水。

——释普济，《五灯会元·卷十七》

忘却是一种能力。

程序员要成长为软件架构师，不得不面对的一个基本问题就是：必须突破 OOP 思维，在进行面向对象设计的时候不要受到过多 OOP 编程语言级细节的干扰。换句话说，就是要有在设计之时将 OOP 细节忘却的能力。

这是因为，在不同层面思考；需要考虑不同的问题。OOD 应更多地将精力放在职责的划分、变化点的隔离、交互机制的确定等问题上，这和 OOP 所关注的不是一回事。反过来讲，不能忘却 OOP 级的编程细节，也就无法真正在 OOD 层面上思考。

封装、继承、多态是 OO 的三大特性，由此可见继承思想的重要性。但是，不少人对继承的理解过多地局限在 OOP 层面，从而限制了继承思想在 OOD 层面的巨大作用。在程序员成长为软件架构师的过程中，必须不断提升对 OO 思想的认识层面，加强实际的设计能力。

本章站在 OOD 的角度，将继承看成实现 OOD 的强大手段，通过具体例子，说明针对接口编程（Program To An Interface）、混入类（Mix In Class）、基于角色的设计（Role-based Design）这三个与继承紧密相关的著名 OOD 技巧。

### 19.1 从一则禅师语录说起

---

《五灯会元》卷十七中，有一则青原惟信禅师的语录：“老僧三十年前未参禅时，见山是山，见水是水。及至后来亲见知识，有个入处，见山不是山，见水不是水。而今得个休歇处，依前见山只是山，见水只是水。”

禅师高论，颇具哲理，讲的是悟道的过程。其实，领悟 OOD 之道的过程又何尝不是如此呢？

### 19.1.1 见继承是继承——程序员境界

初学 OOP 的人，大多处在“见继承是继承”的层面，最关心的是类的语法、类的成员变量、类的成员函数等这些实现层的东西。这是程序员境界。

### 19.1.2 见继承不是继承——成长境界

开始研习 OOD 之时，又往往跳到另一个极端，只关心设计，而无心（也可能是无力）关心实现，处在所谓“见继承不是继承”的层面。在这个阶段的人，脑中的兴奋点是“设计”，是职责分配、接口设计、可重用性、可扩展性、耦合度、聚合度等这些设计层的概念。这是成长境界。

### 19.1.3 见继承只是继承——设计师境界

学通 OOD 之后，会达到“见继承只是继承”的层面。一个“只”字，体现了继承背后的“设计理念”才是该境界的要害。但是，这个阶段和第二阶段不同，第二阶段是一味的否定，而本阶段是否定之否定，把 OOP 层面的继承机制看成用来实现特定 OOD 的手段加以利用。这是设计师境界。

## 19.2 从 OOD 层面认识继承

在 OOP 层面，除了类、成员变量、成员函数这些最基本的概念，最重要的就是代码重用和名字空间的可见性了。而在 OOD 层面，最基本的概念是类、职责、状态、角色这些更抽象一级的概念，及其相关的耦合度、聚合度、可重用性、可扩展性、可维护性等。可见，虽然 OOD 最终要依赖 OOP 作为实现手段，但显然 OOD 和 OOP 并非在同一抽象级上，它们分别有不同的概念体系和思维重点。

再说继承。单纯从 OOP 层面看，继承是一个通过复用父类功能而扩展应用功能的基本机制，它允许你根据旧的类快速定义新的类；还有些人用继承仅为了获取名字空间的可访问性。但是，从 OOD 层面看，继承可以演变出“Is-A”、“Plays Role Of”等抽象的设计概念。因此，担任设计师角色的人如果自己还限制在 OOP 的层面，“设计乏术”的局面是不可避免的。总之，提升对继承的认识，对通过活用接口继承和实现继承这两种机制来实现 OOD 非常重要。

与继承相关的 OOD 技巧有很多，本章仅讨论针对接口编程、混入类、基于角色的设计这三种技巧，图 19-1 展示了它们和继承的关系。

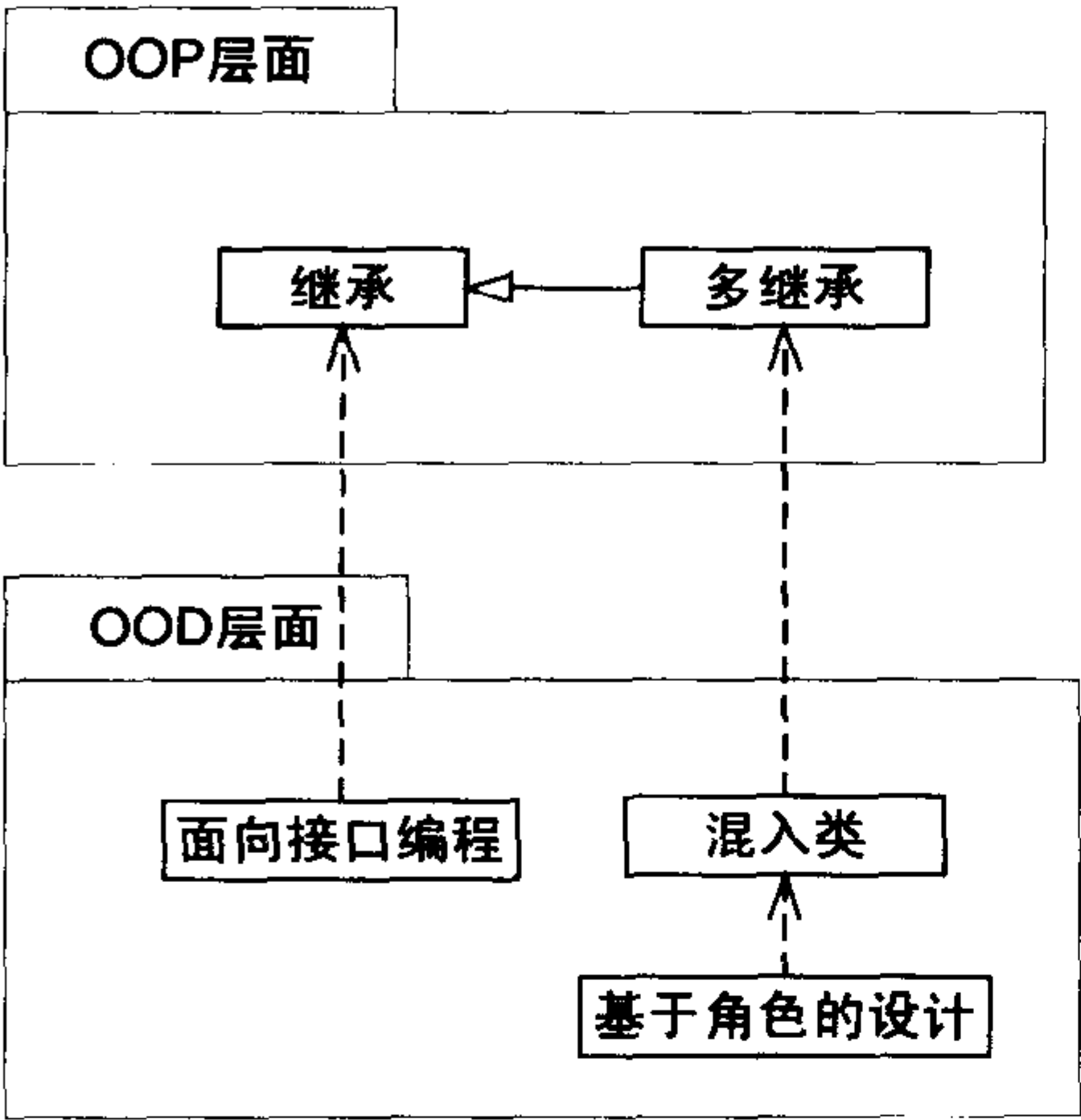


图 19-1 继承的 OOD 应用

## 19.3 针对接口编程——隔离变化

### 19.3.1 相关理论

耦合是依赖的同义词，被定义为“两个元素之间的一种关系，其中一个元素变化，导致另一个元素变化”。抽象耦合被定义为“若类 A 维护一个指向抽象类 B 的引用，则称类 A 抽象耦合于 B”。

依赖性倒置原则（Dependency Inversion Principle）形式化了抽象耦合的概念，明确表述了应该“依赖于抽象类，不要依赖于具体类”。

针对接口编程遵守上述原则，从而在很大程度上阻止了变化波及范围的扩大，有效地隔离了变化，有助于增强系统的可重用性和可扩展性。

### 19.3.2 针对接口编程举例——用于架构设计

根据经典的 Coad 的 OOD 理论，一个软件系统通常包含四层：用户界面层、问题领域层、数据管理层、系统交互层，如图 19-2 所示。

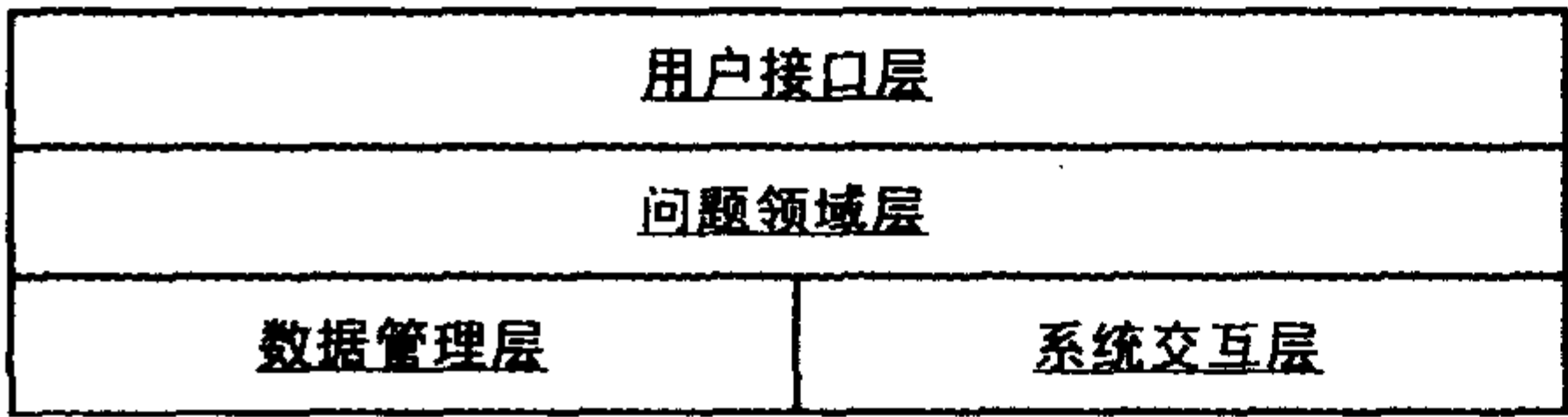


图 19-2 四层架构

将架构划分为层的一个很大好处是，这些层形成了开发小组的自然分界——每层的开发人员所需要的技巧是不同的。用户界面层的开发小组需要了解将使用的用户界面工具包；数据管理层的开发小组需要熟悉相关的数据库、持久工具或者使用的文件系统；系统交互层的开发小组需要了解通讯协议和用到的中间件产品；问题领域层的开发小组不需要了解这些知识，他们需要最深入的领域知识，以及用到的相关分布对象或组件技术。

但是，要真正使得各个开发小组最大限度地独立开发，还需要一个稳定的架构设计做保证才行，其设计的核心思想是：问题领域层“不依赖于”其他任何层，而其他任何层“只依赖于”问题领域层。如图 19-3 所示。

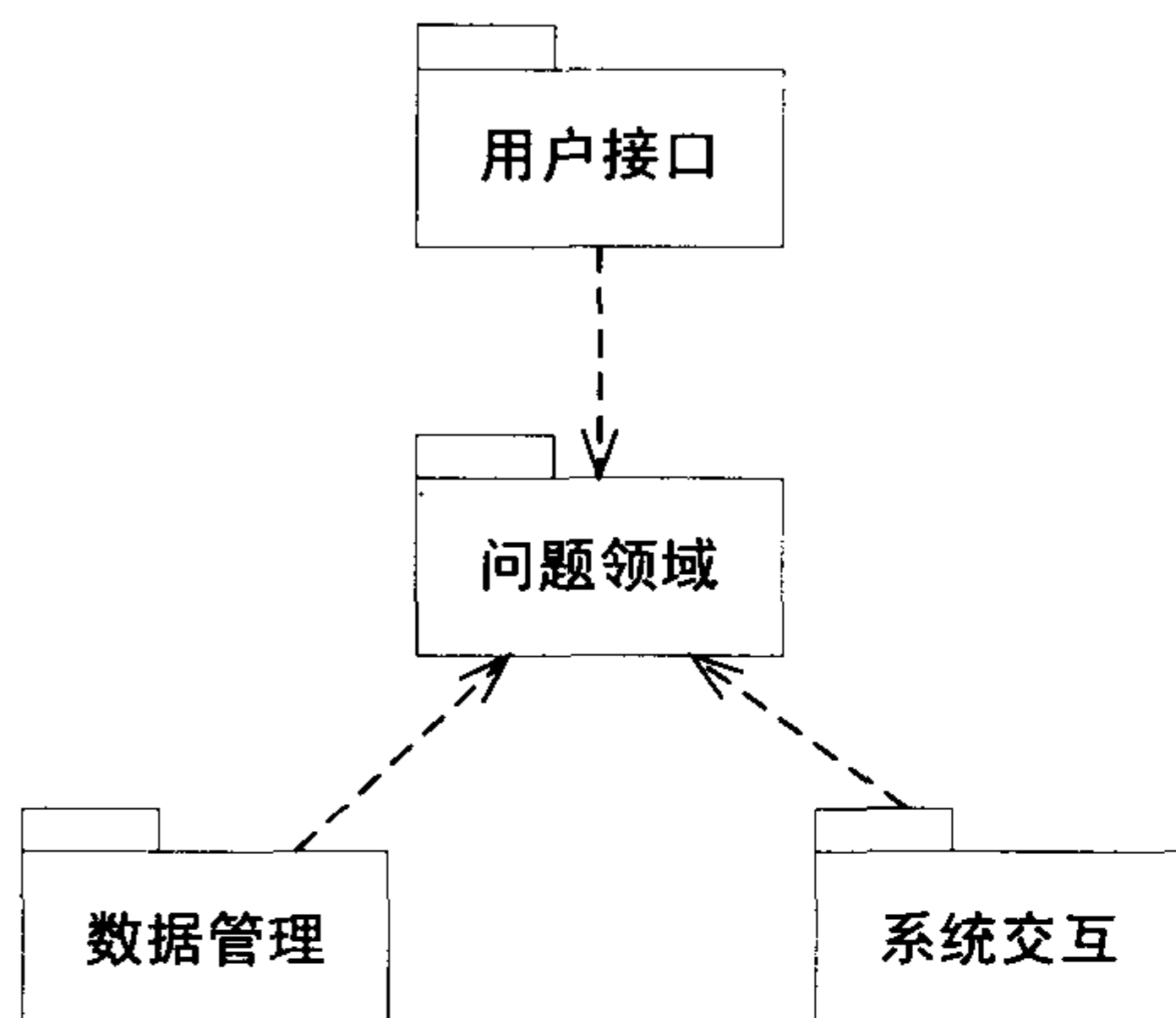


图 19-3 层之间的理想依赖关系

该架构设计的实现，极为重要的一点，就是要使用针对接口编程的技巧。以系统交互层对问题领域层的单向依赖为例：

- 如果系统交互层要调用问题领域层的操作，直接调用即可。
- 如果问题领域层要调用系统交互层的操作，需要由问题领域小组定义一个通用的抽象接口，通过针对接口编程调用这个抽象接口；而系统交互小组通过接口继承机制，定义抽象接口的子类，该子类完成抽象接口的具体实现。

笔者曾有一个项目，该系统需要实时地将本系统的数据变化，通知远端的另一个系统。相关设计如图 19-4 所示。在问题领域层，仅包含了一个抽象接口 `CChangeReporter`，而并不关心 `CChangeReporter` 的具体实现。系统交互层拥有选择具体实现方法的自由，比如 `CSoapChangeReporter` 是用 SOAP 通讯协议实现的 `CChangeReporter`，`CTcpChangeReporter` 是用 TCP 协议实现的 `CChangeReporter`。而且如果由于技术的或商业的原因，将来需要同时支持多种通讯协议，也比较容易。



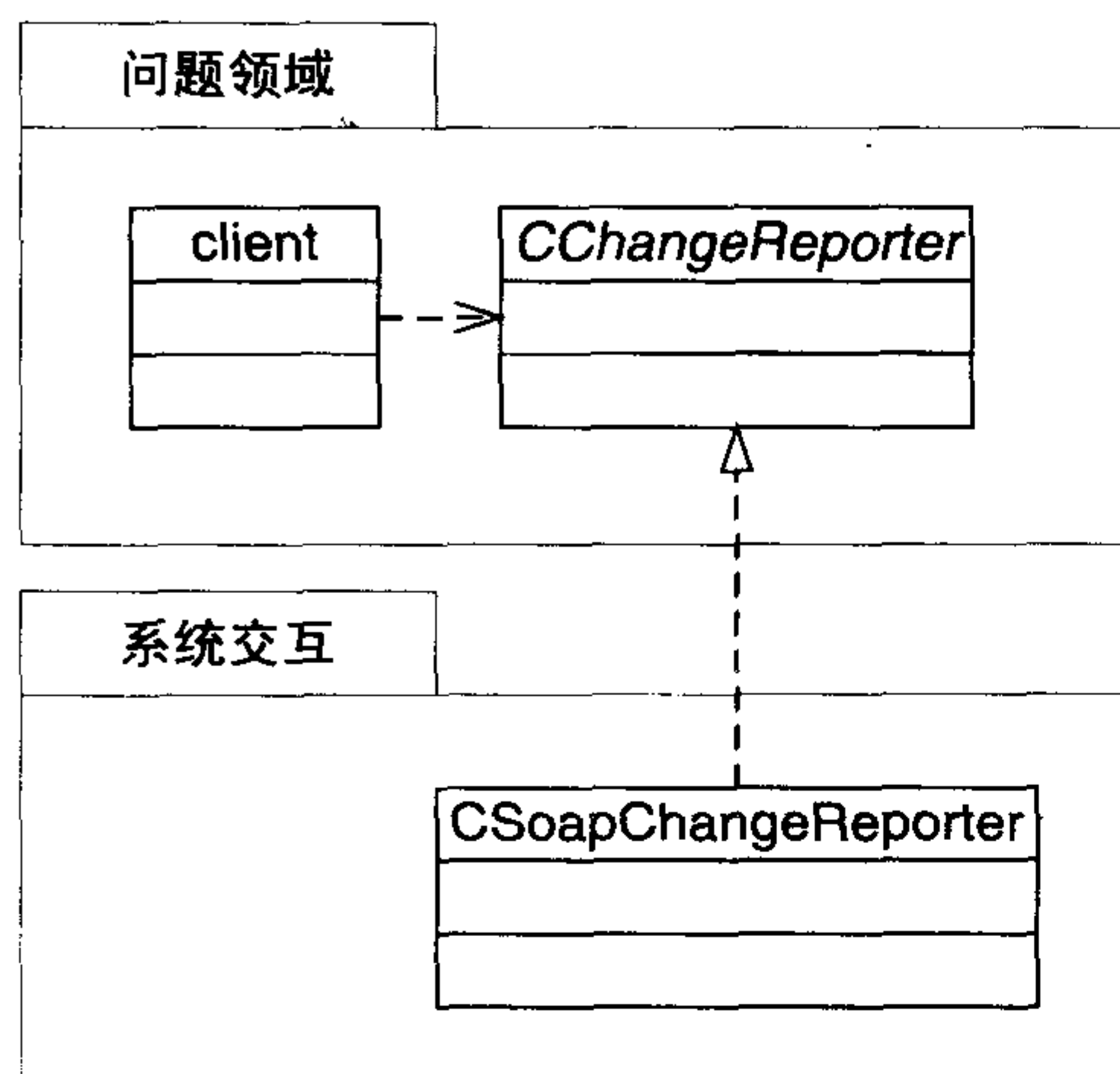


图 19-4 利用针对接口编程来处理依赖方向

### 19.3.3 针对接口编程举例——用于类设计

上一章中，我们详述了如何使用策略模式来设计一个可重用、易扩充的 MIME 类层次（如图 19-5 所示），其中抽象接口类 CMimeAlgo 起到了至关重要的作用。

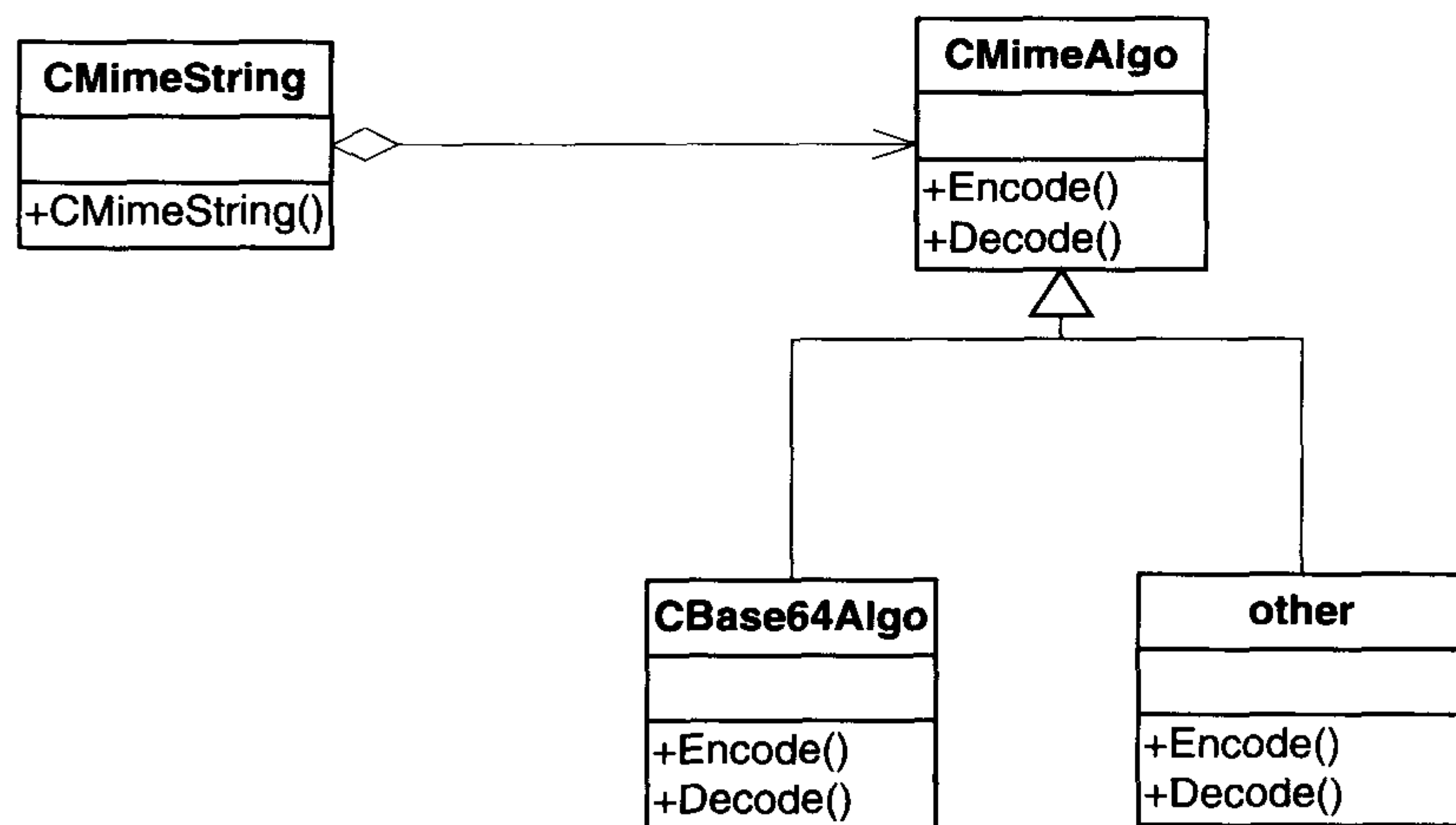


图 19-5 针对接口编程是这个设计的核心

针对接口编程是这个设计的核心。用户通过 **CMimeString** 使用 MIME 编码的功能，**CMimeString** 允许用户在运行过程中动态配置 MIME 编码的具体算法；具体 MIME 编码算法由 **CMimeAlgo** 类层次提供，具体的 **CMimeAlgo** 子类的实例化是由 **CMimeString** 根据用户的配置动态完成的；要增加新的 MIME 编码算法，只需实现新的 **CMimeAlgo** 子类，并简单扩充 **CMimeString** 的动态实例化代码即可。

## 19.4 混入类——更好的重用性

### 19.4.1 相关理论

混入类被定义为“一种被设计为通过继承与其他类结合的类”，它给其他类提供可选择的接口或功能。

从实现上讲，混入类要求多继承；混入类通常是抽象类，不能实例化。

混入类的作用在于：它不仅可以提高功能的重用性，减小代码冗余；而且还可以使相关的“行为”集中在一个类中，而不是分布到多个类中，避免了所谓的“代码分散”和“代码交织”问题，提高了可维护性。

### 19.4.2 混入类举例

来看一个具体项目。在一个信用卡客户服务系统项目中，要求能够以多种方式发送多种信息给用户，并能够适应未来业务的发展变化。

当前系统需要支持的发送方式：

- 打印（并邮寄）
- E-mail
- 传真

可预见的未来要支持的发送方式：

- 手机短信
- PDA 消息

当前系统需要支持的待发送信息：

- 信用卡对账单
- 信用卡透支催收单

可预见的未来要支持的待发送信息：

- 信用卡新业务宣传单
- 信用卡促销活动宣传单

下面是一些设计考虑。一种发送方式要支持多种待发送信息，我们希望发送功能有很好的可重用性；为了便于将来支持新的发送方式和发送信息，设计必须具有良好的可扩展性。相关设计如图 19-6 所示。其中采用了混入类的 OOD 技巧，用一个 CSendableDoc 作为混入类，支持发送功能的重用；CSendableDoc 还采用了策略模式支持发送方式的扩充。

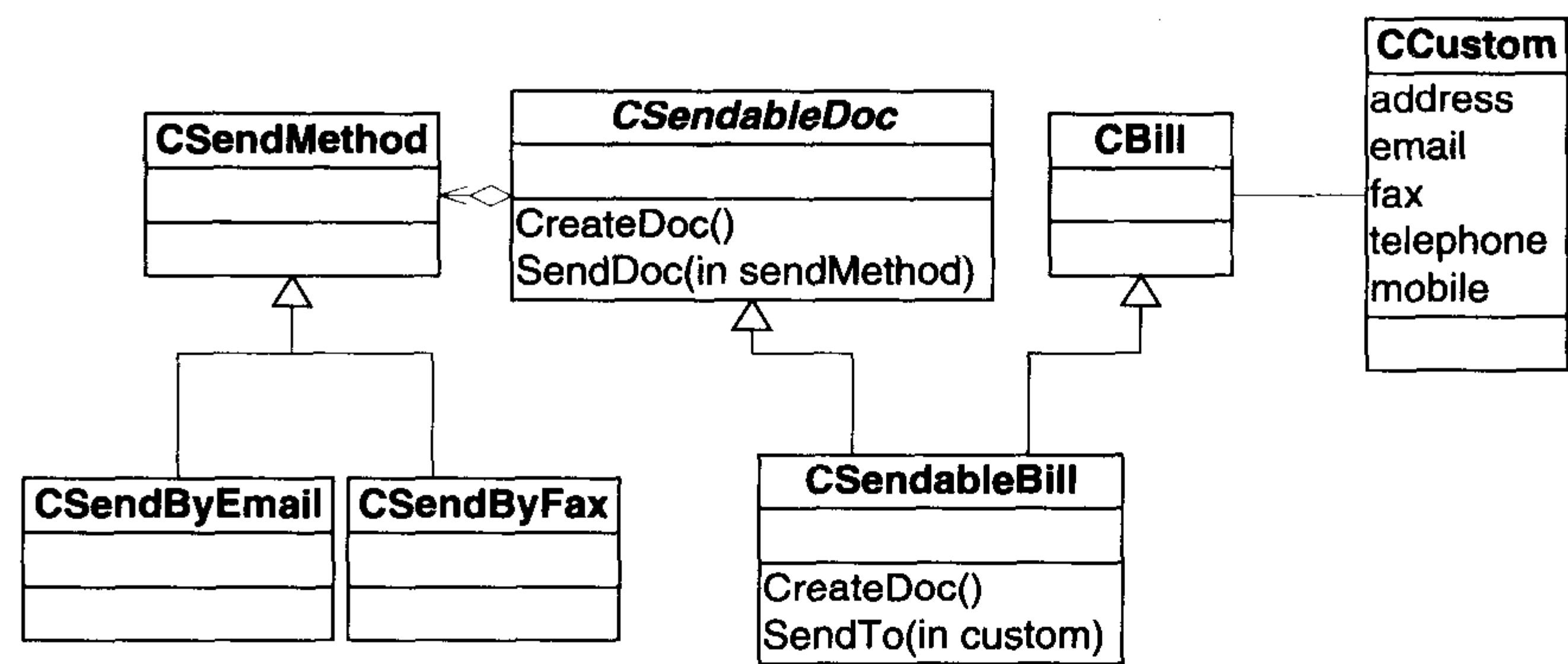


图 19-6 一个采用混入类的设计

## 19.5 基于角色的设计——使用角色组装协作

### 19.5.1 相关理论

协作被定义为“多个对象为了完成某种目标而进行的交互”。而角色是“特定协作中的对象的抽象”，它“仅定义了对对象特征的一个对某协作有意义的子集”。协作和角色的概念和现实世界很接近，图 19-7 的例子展示了角色在协作中的作用。由例子可以看出，基于角色的设计的意义在于：我们很容易通过已有角色的组合来构造新的协作，以完成新的功能。

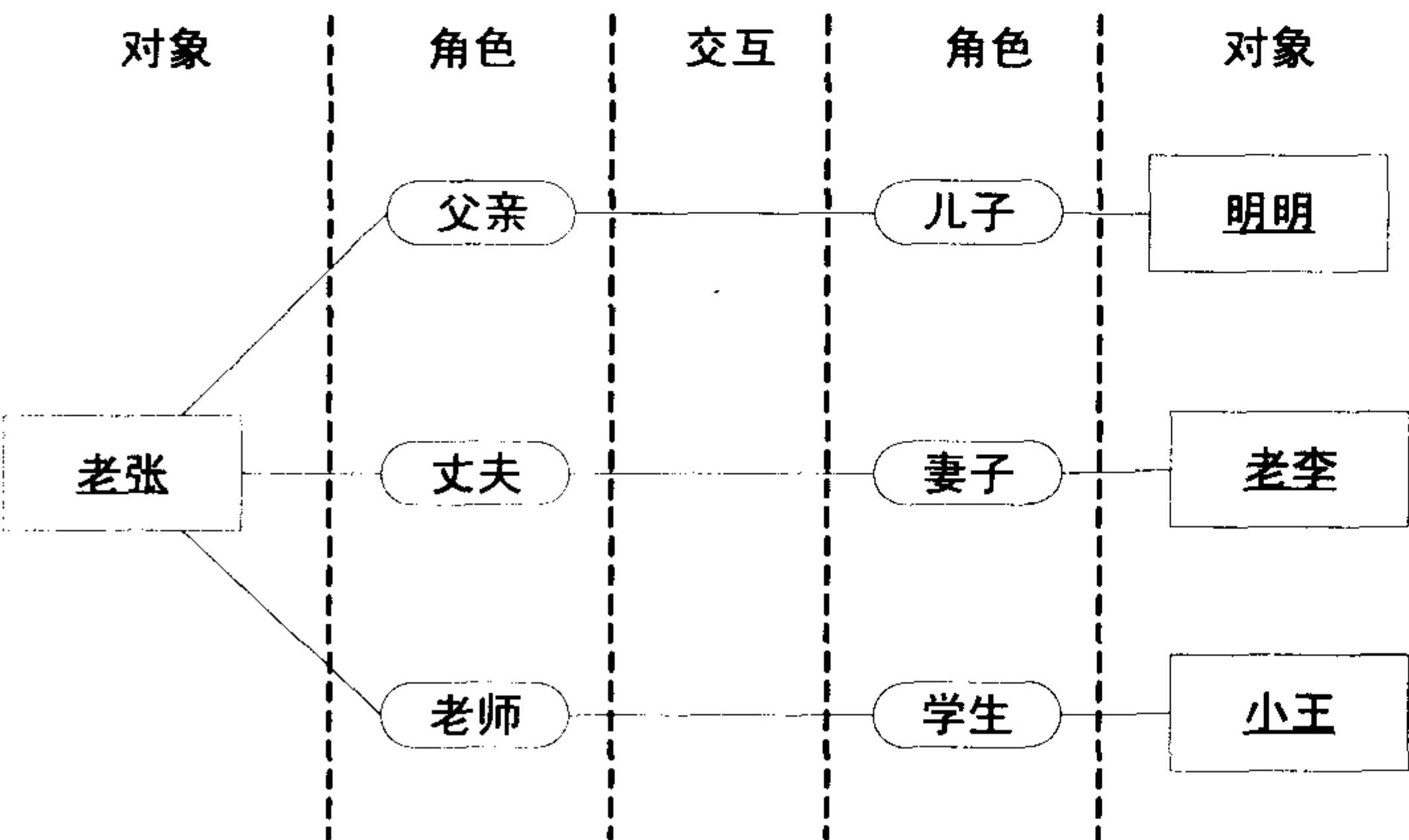


图 19-7 角色与协作

接口隔离原则（Interface Separation Principle）信奉“多个专用接口优于一个单一的通用接口”的思想，因为“任何接口都应当具有高内聚性”，以便“保证实现该接口的类的实例对象可以只呈现为单一的角色”。

## 19.5.2 基于角色的设计举例

比如，待开发的一个软件系统，其后台数据源可能是关系数据库、一般的文件，还可能是另一个私有数据库。既然接口可以隔离变化，我们可以定义一个单一的接口，为所有的数据客户类提供服务。如图 19-8 所示。

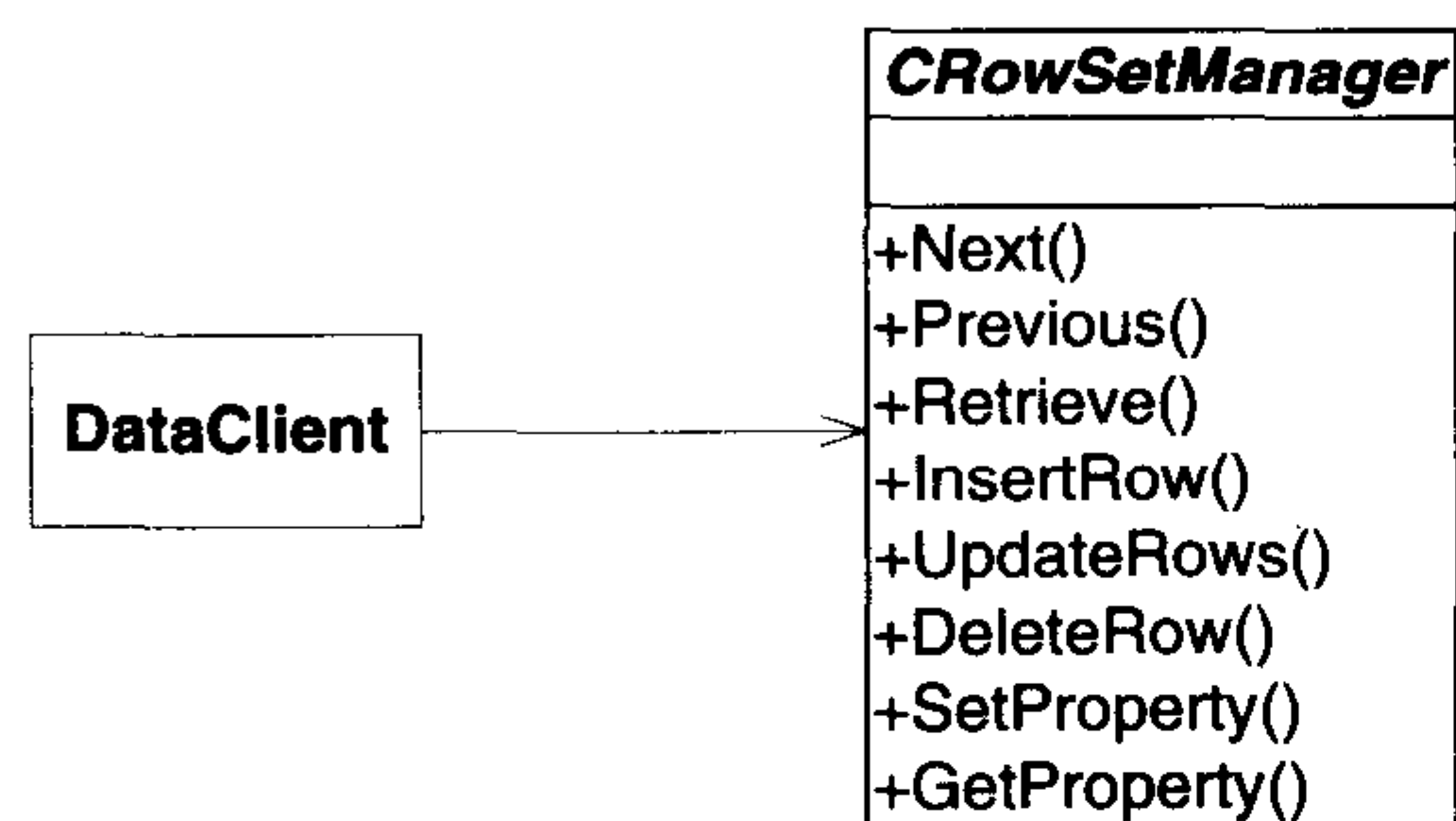


图 19-8 “宽”接口设计

但是，上面的设计违背了基于角色的设计思想，根本不能保证“实现该接口的类的实例对象可以只呈现为单一的角色”，这会带来一些问题。比如，有一个数据客户类，不需要插入、更新等功能，而仅仅需要对数据进行读操作，这时显然一个提供“读”服务的“角色”是最合理的设计，但 CRowSetManager 却是如此之“宽”的一个接口。最终，我们可以这样来改进设计，如图 19-9 所示。

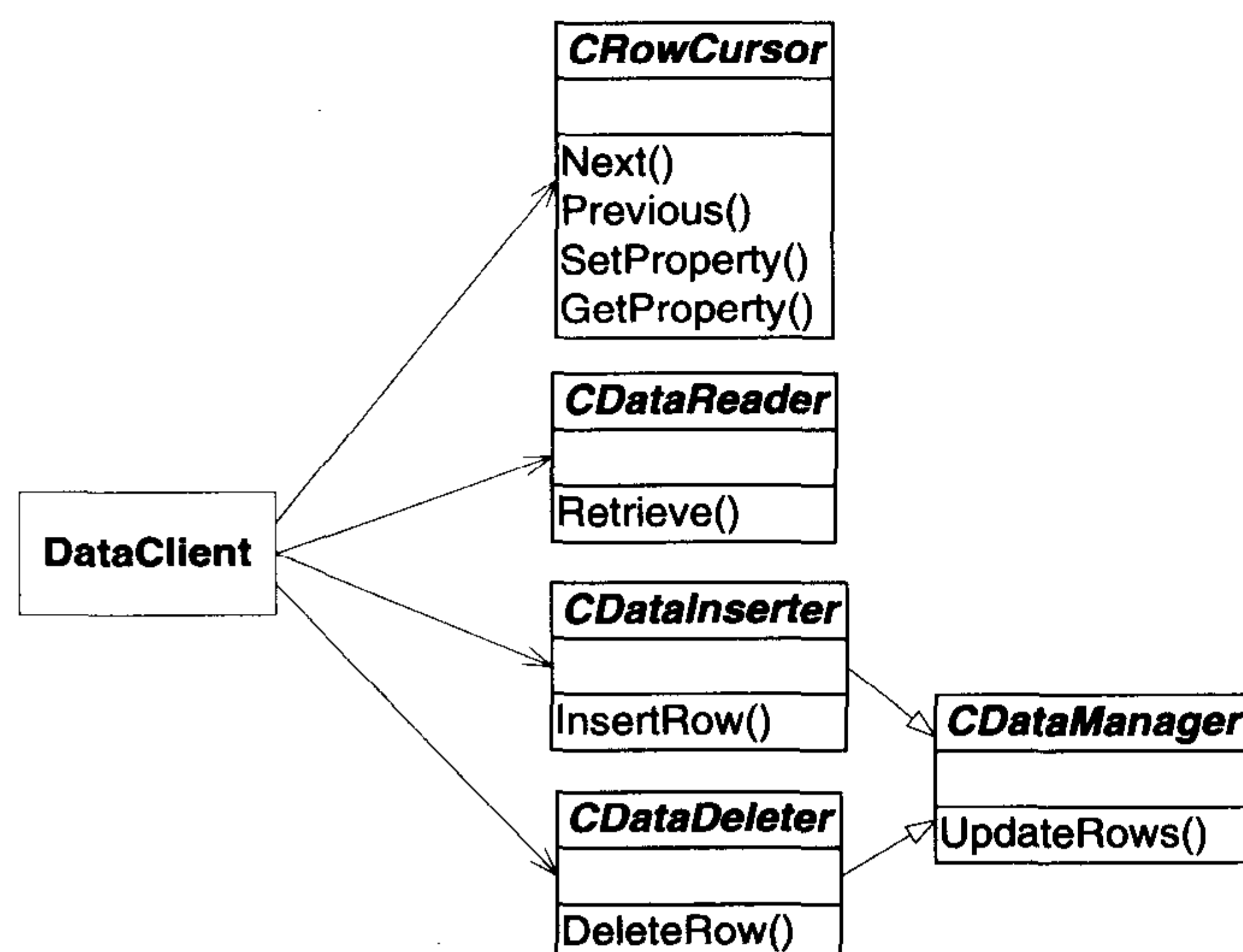


图 19-9 运用基于角色的设计进行接口隔离



## 第 20 章 细微见真章：耦合其实并不空洞

---

对软件设计者来说，被简单、直观地分割，并具有最小内部耦合的内部结构就是美的。

——Robert C. Martin, 《软件之美》

曾听到过这样的说法：“软件架构师很好识别，他们是那些开会时言必称‘耦合’的人。”虽是开玩笑，但言下之意是指有些架构师常空谈“耦合”，其实未必能将耦合落到实处。

本章将结合代码，谈谈耦合性对设计的影响。

### 20.1 顺序耦合性简介

---

耦合性（Coupling）是两个元素之间的一种关系，为了保持正确性，需要两个元素同时进行变化。耦合性可以分为**静态耦合性**、**动态耦合性**和**差异耦合性**三大类。

顺序耦合性是静态耦合性的一种，为了保持正确性，两个元素必须以特定的顺序出现。

例如，下列两条语句的顺序，反映了变量“先定义后使用”的要求，不能颠倒。

```
int i;  
i = 6 ;
```

再例如，下列两条语句的顺序，也不能颠倒。

```
FILE * fp = fopen( "a.txt" );  
fclose( fp );
```

### 20.2 案例研究：顺序耦合性 Bug 一例

---

本节讲述笔者在实际工作中遇到的一个“顺序耦合性 Bug”，这种 Bug 通过代码走查很难发现，常常要借助跟踪调试才行；最后通过重构改进设计，从根本上解除顺序耦合性隐患。

20.2.1 项目简介

这是一个多媒体编辑软件，要求编辑的结果可以保存为“项目”，便于以后打开或进一步编辑。经简化的软件架构如图 20-1 所示。问题领域层（PD）对多媒体编辑领域进行了抽象，数据管理层（DM）负责数据的保存和读取，用户界面层（UI）负责为用户提供图形化用户界面。并且，DM 层和 UI 层并不相互依赖，它们都仅依赖于 PD 层，这样的软件架构将 PD 层、DM 层和 UI 层最大限度地独立开来，有利于应对需求的变更。很自然地，由 3 个相对独立的开发小组，分别负责上述 3 个子系统。

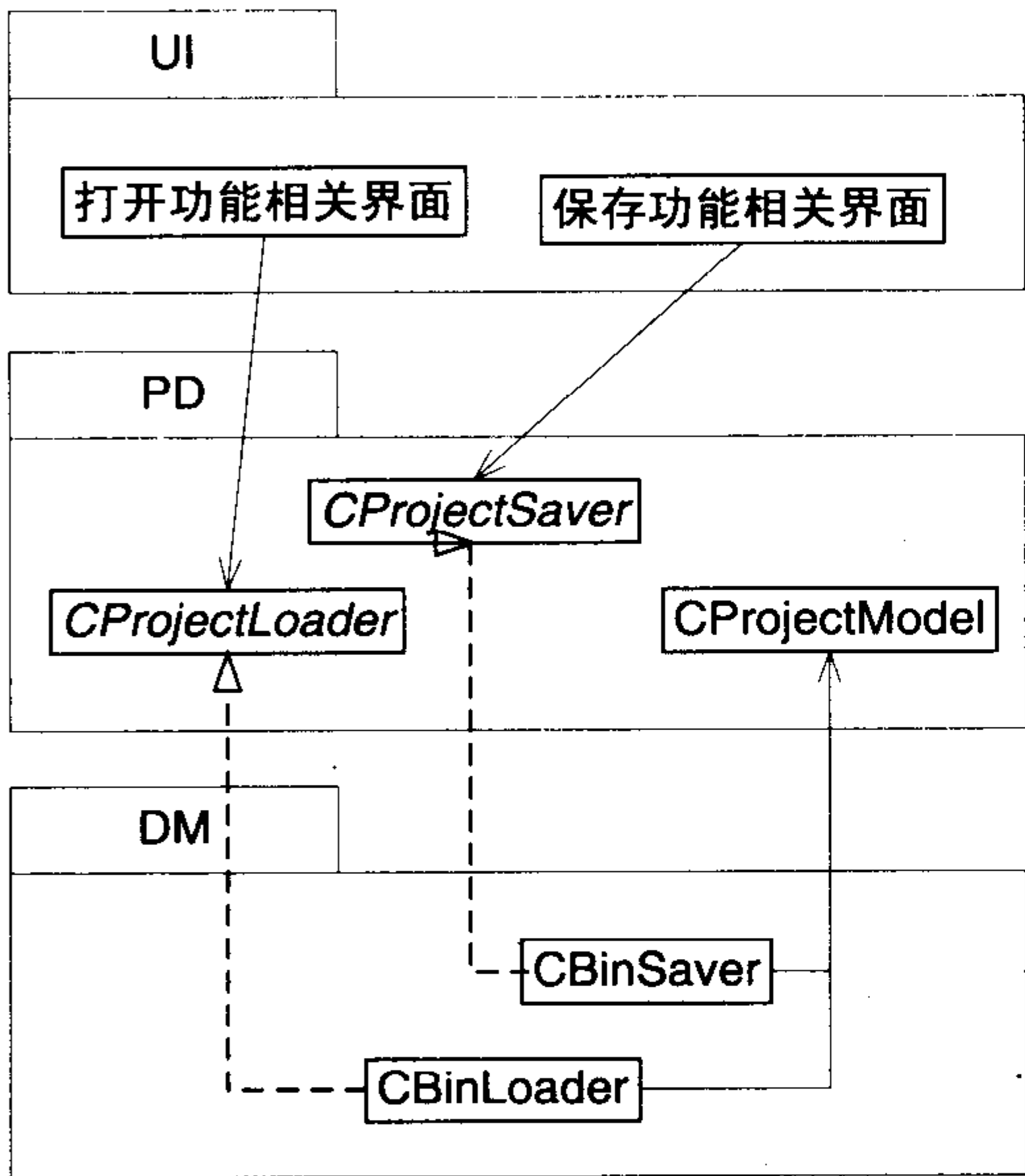


图 20-1 该多媒体编辑软件的简化架构

20.2.2 新的需求

原先版本的保存和打开功能，都是针对专有格式的二进制“项目”文件。在新的版本中，要求支持 XML 格式的“项目”文件。

由于软件架构设计的合理性，支持这项新需求的工作相当简单。首先，PD 小组和 DM 小组一起，制定 XML 文件的格式，形成文档。然后，DM 小组指派 XML 高手，写一个叫做 CXmlSaver 的类去实现 CProjectSaver 接口，该类按照 XML 文件格式文档，将 Project Model 保存到 XML 中去；同理，还要写一个叫做 CXmlLoader 的类去实现 CProjectLoader 接口；如图 20-2 所示。最后，UI 小组做少量工作，增加相关功能的界面。

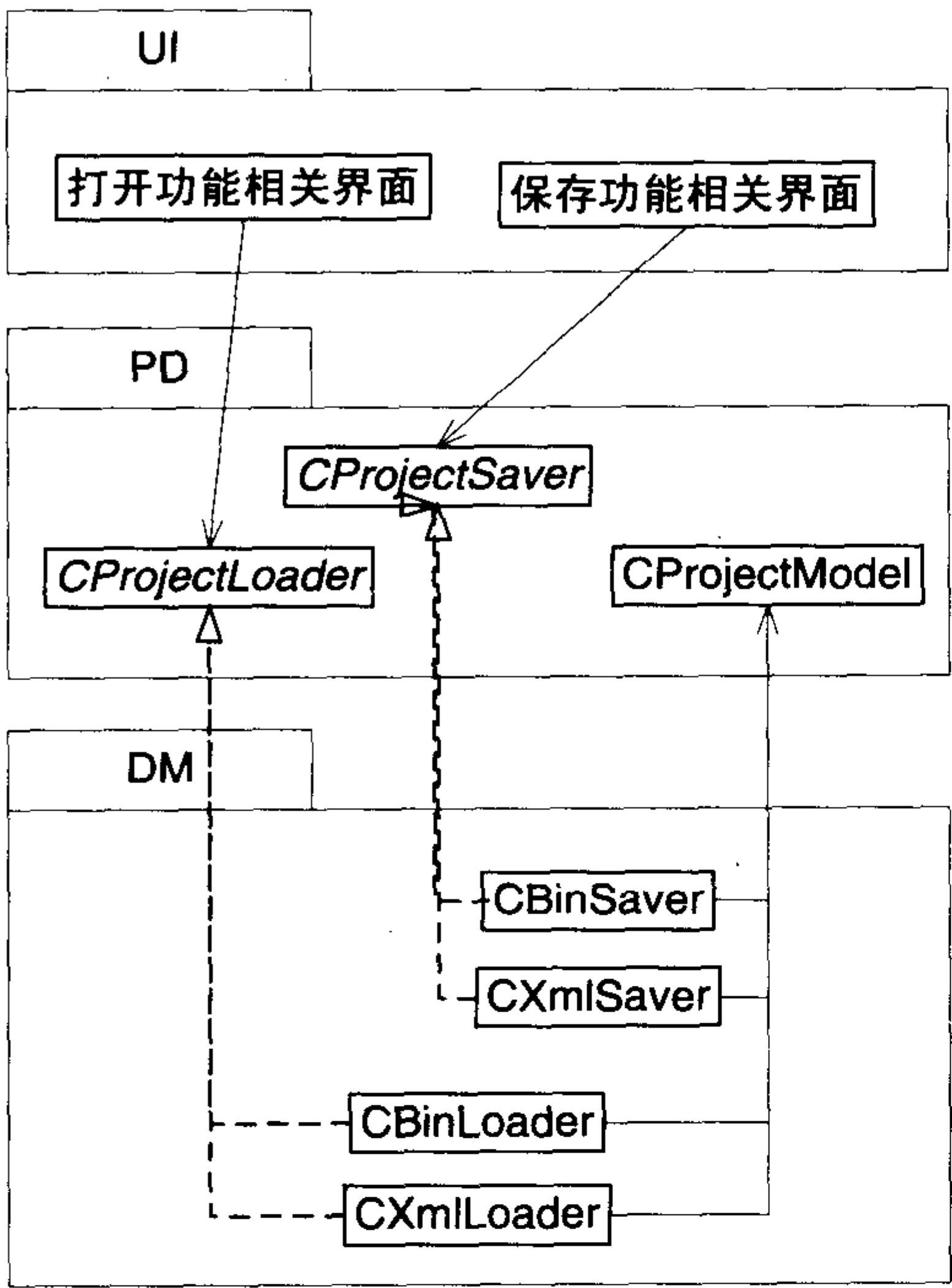


图 20-2 扩充后的设计

20.2.3 发现顺序耦合性 Bug

哈，我就是 DM 小组中被指派写 CXmlLoader 的家伙。但是很不幸，我的下列代码有 Bug，不能将 XML 文件中的 Video File 的信息正确地 Load 到 Project Model 中的 Video File 节点。

```
void CXmlLoader::LoadVideoFile(CVideoFile * inVideo)
{
    ...
    inVideo->SetStartTime( GetNodeAttr(Video, "startTime") );
    inVideo->SetEndTime( GetNodeAttr(Video, "endTime") );
    inVideo->SetDuration( GetNodeAttr(Video, "duration") );
    ...
}
```

20.2.4 跟踪调试

这到底是怎样回事呢？我进行了一番代码走查，并没有发现什么不合理的地方。

于是，我在 inVideo->SetEndTime( GetNodeAttr(Video, "endTime") );处设置断点，然后跟踪到 void CVideoFile::SetEndTime(double inEndTime)里面去。我看到了下面的代码。

```
CVideoFile::CVideoFile()
{
    ...
    mDuration = 0;
    ...
}
void CVideoFile::SetEndTime(double inEndTime)
{
    if (inEndTime > mDuration)
        mEndTime = mDuration;
    else if (inEndTime > 0)
        mEndTime = inEndTime;
    else
        mEndTime = 0;
}
```

查看一下变量当前的值。呀，此处的 `mDuration` 还保持着构造函数中为它赋的初值 0 哟，而 `inEndTime` 的值是 324.6。于是，`if (inEndTime > mDuration)` 成立，所以 `mEndTime` 被设置为 `mDuration` 的值 0。

再跟踪到 `inVideo->SetDuration( GetNodeAttr(Video, "duration") );` 里面看看，我看到了下面的代码。

```
void CVideoFile::SetDuration(double inDuration)
{
    if (inDuration > 0)
        mDuration = inDuration;
    else
        mDuration = 0;
}
```

看一下变量当前的值。咦，这里 `inDuration` 的值是 600.0，倒是比刚才要设置的 `EndTime` 的值 324.6 大，显然，刚才 `inVideo->SetEndTime( GetNodeAttr(Video, "endTime") );` 中，应该用这个 `Duration` 值来进行合法性检查的。喔，调用 `inVideo->SetDuration( GetNodeAttr(Video, "duration") );` 晚了，应该在 `inVideo->SetEndTime( GetNodeAttr(Video, "endTime") );` 之前就调用的。

原因找到了，只需调整一下语句的顺序，Bug 就被修正了。代码如下所示。

```
void CXmlLoader::LoadVideoFile(CVideoFile * inVideo)
{
    ...
    inVideo->SetDuration( GetNodeAttr(Video, "duration") );
    inVideo->SetStartTime( GetNodeAttr(Video, "startTime") );
    inVideo->SetEndTime( GetNodeAttr(Video, "endTime") );
    ...
}
```



### 20.2.5 分析原因

先来分析一下 CVideoFile 的设计思想。作为 Project Model 的一部分，PD 小组实现了一个严格检查合法性的 CVideoFile，任何时候都不允许一个 Video File 的结束时间比其总时间大。

再看问题出在哪里？从 CVideoFile 本身提供的接口，看不出顺序耦合性的存在——我无从知道必须先调用 CVideoFile::SetDuration()，后调用 CVideoFile::SetEndTime()。

简言之，由于没有处理好耦合性，因此 CVideoFile 的接口设计并不合理。

### 20.2.6 解决策略

从表面上看，Bug 已经被修正。但是，顺序耦合性的隐患还在；一不留神，同样的 Bug 还会出现。下面，我们通过重构改进设计，从根本上解除顺序耦合性隐患。

设计一个进行严格检查合法性的 CVideoFile，是合理的，应当保留。

但是 CVideoFile 的接口设计的不合理。我们应当改进 CVideoFile 的接口设计，消除其中的顺序耦合性隐患，使用户在不知道先设置 Duration 后设置 EndTime 的情况下，也不会犯错误。

下面我们通过一步步地重构，来达到这个目的。

### 20.2.7 运用重构的“Extract Method”成例

重构的“Extract Method”成例建议，如果一个代码段拥有完整的意义，则应当将它们单独抽出，放入一个方法，并根据其意义为方法命名。

下列使用 CVideoFile 的代码段，正是这样的情况；而且，如果不改进，很容易出现顺序耦合性 Bug。

```
void CXmlLoader::LoadVideoFile(CVideoFile * inVideo)
{
    ...
    inVideo->SetDuration( GetNodeAttr(Video, "duration") );
    inVideo->SetStartTime( GetNodeAttr(Video, "startTime") );
    inVideo->SetEndTime( GetNodeAttr(Video, "endTime") );
    ...
}
```

但是，我们决定对 CVideoFile 本身进行“Extract Method”重构，而不是对 CVideoFile 的客户代码，因为这样所有使用 CVideoFile 的客户代码都可以从中受益。我们为 CVideoFile 增加了这样一个公共方法：

```
void CVideoFile::SetTribbleTime(double inStartTime,
    double inEndTime, double inDuration)
{
    SetDuration(inDuration);
```

```
SetStartTime(inStartTime);  
SetEndTime(inEndTime);  
}
```

### 20.2.8 运用重构的“Hide Method”成例

重构的“Hide Method”成例建议，如果一个方法并不被除了所属类之外的其他类使用，那么该方法应当是 `private` 方法。

作为 `CVideoFile` 的设计者，我希望客户程序调用 `SetTribbleTime()` 去设置相关属性，并不希望客户程序直接调用 `SetDuration()`、`SetStartTime()` 和 `SetEndTime()` 中的任何一个。所以我将这 3 个方法置为 `private` 的。

### 20.2.9 运用重构的“Introduce Parameter Object”成例

重构的“Introduce Parameter Object”成例建议，如果一个方法的多个参数可以很自然地划分在一起，就应当用一个对象来代替它们。

方法 `void CVideoFile::SetTribbleTime(double inStartTime, double inEndTime, double inDuration)` 的 3 个参数显然满足这样的条件，所以，我们可以引入一个 `CTribbleTime` 类来代替它们，代码如下。

```
class CTribbleTime  
{  
    double duration;  
    double startTime;  
    double endTime;  
};  
void CVideoFile::SetTribbleTime(CTribbleTime & inTribbleTime)  
{  
    SetDuration(inTribbleTime.duration);  
    SetStartTime(inTribbleTime.startTime);  
    SetEndTime(inTribbleTime.endTime);  
}
```

### 20.2.10 其他改进

现在的设计并不完美，因为我们在进行多媒体编辑的时候，可能只需要设置 `Start Time`，这时代码会像下面所示，很繁琐。

```
CTribbleTime t;  
t.startTime = 200;  
t.endTime = video.GetEndTime();  
t.duration = video.GetDuration();  
video.SetTribbleTime( t );
```

下面，我们为 CVideoFile 增加一个 GetTribleTime()方法，通过它可以方便地取得 CVideoFile 的 3 个 Time 的当前值。最后，我们仅设置 Start Time 的代码看起来像这样。

```
CTribleTime t = video.GetTribleTime();
t.startTime = 200;
video.SetTribleTime( t );
```

另外，我们可以将 CVideoFile 的 3 个成员变量 mStartTime、mEndTime 和 mDuration 改用一个 CTribleTime mTribleTime 实现；删除 GetDuration()、GetStartTime()和 GetEndTime()这 3 个方法；等等。如图 20-3 所示。

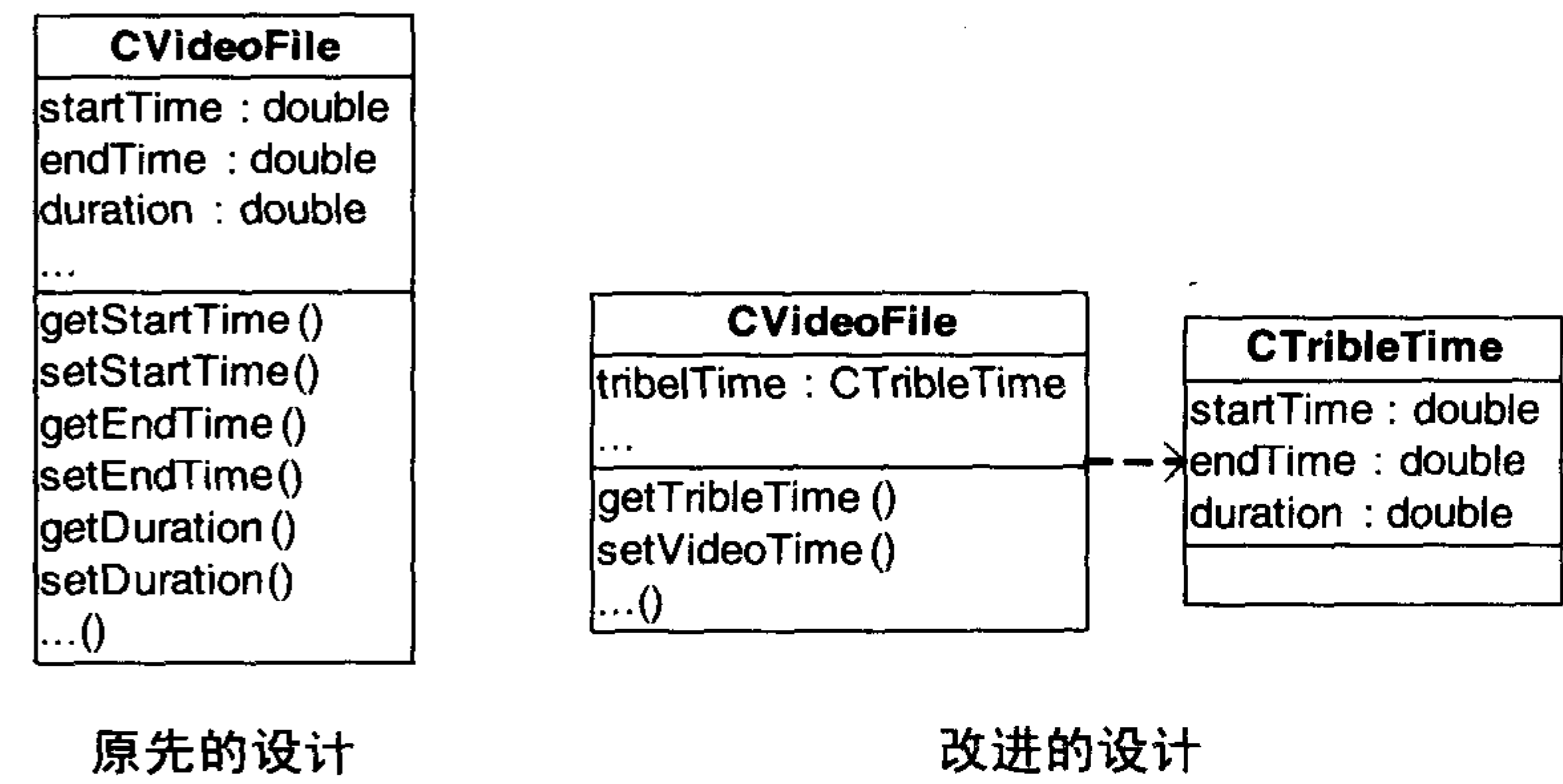


图 20-3 改进的 CVideoFile 设计





## 第 21 章 敏捷设计：从理论到实践

---

未来将越来越不可预测，这是新经济最具挑战性的方面之一。商务和技术上的瞬息万变会产生变化，这既可以看作要防范的威胁，也可以看作应该欢迎的机遇。

——Martin Fowler & Jim Highsmith, 《敏捷宣言》

如何应付软件开发与维护中的“变化”，一直是近年来备受软件企业关注的问题。敏捷方法的兴起，更是为“按需应变”带来了一股强劲的浪潮。

作为软件架构师，你应该对“将来的软件能在多大程度上适应新的变化”负责。同时，软件架构师又不能为了支持变化而“穷兵黩武”——即支持所有能想象得到的变化，而不管这些变化发生的几率到底有多大，也不管这样做的代价有多大。对此，Robert Martin 在《敏捷软件开发》中曾告诫我们说：

对于应用程序中的每个部分都肆意地进行抽象同样不是一个好主意。正确的做法是，开发人员应该仅仅对程序中呈现出频繁变化的那些部分做出抽象。拒绝不成熟的抽象和抽象本身一样重要。

本章从理论和实践两方面，和大家分享笔者在敏捷设计方面的心得：

- 首先，以一种全新的角度考察耦合，并将其表述为良性依赖原则；
- 然后，通过应用实例，说明该原则如何和著名的“面向对象设计 5 大原则”结合，来“务实地应付变化”；
- 最后，从应付变化的角度，对各原则做综合总结。

本章是“理论联系实际”、务实运用耦合思想的范例。需要说明的是，本章采用“良性依赖原则”的叫法，是出于和依赖倒置原则（Dependency-Inversion Principle）的叫法保持一致的目的；由于“耦合”和“依赖”是一对使用都非常广泛的同义词，所以叫做“良性耦合原则”也是可以的。

## 21.1 换个角度考察依赖

---

### 21.1.1 依赖的概念

依赖 (Dependency)：两个元素之间的一种关系，其中一个元素变化，导致另一个元素变化。

依赖的同义词：耦合 (Coupling)，共生 (Connascence)。

依赖的危害：如果被依赖元素发生变化，可能引起另一个元素不得不变化。

### 21.1.2 从会不会造成“实际危害”的角度考察依赖

关于依赖，已经研究得很多了：从依赖程度的大小角度来考察，有了耦合度相关理论；从依赖产生的原因角度来考察，有了静态共生性、动态共生性和差异共生性的相关理论；不一而足。

是的，如果被依赖元素发生变化，可能引起另一个元素不得不变化，这就是依赖的危害。但是，如果被依赖元素不发生变化呢？答案是：不会造成危害！于是，“冤案”产生了：由于需求分析上的偏差，设计中“在理论上”很稳定的耦合度低的依赖，可能“在实际中”恰恰是给我们造成危害的家伙；相反，“在理论上”声名狼藉的耦合度高的依赖，“在实际中”也可能并不给我们造成任何危害。

于是，我们很自然地想到，区分依赖的“实际危害”和“理论危害”是有实践意义的。下面，从会不会造成“实际危害”的角度来考察依赖，将其分为良性依赖和恶性依赖两种类型。

- 恶性依赖：被依赖的元素“在实际中”，而不是“在理论上”，是“易变的”；
- 良性依赖：被依赖的元素“在实际中”，而不是“在理论上”，是“不易变的”。

## 21.2 良性依赖原则

---

不会“在实际中”造成危害的依赖关系，都是良性依赖；依赖的“理论危害”不一定成为“实际危害”，反之亦然。这就是良性依赖原则。

依赖是不可避免的，重要的是如何务实地应付变化。这就是良性依赖原则要做的。

### 21.2.1 依赖是不可避免的

OOD 的实质，简而言之，就是妥善地为多个类进行职责分配，使这些类相互协作而构造出完成特定功能的系统。在软件设计中，依赖是不可避免的，就像人类社会不可能没有人与人之间的协作与依赖一样，这一点其实是不言自明的。

### 21.2.2 重要的是如何务实地应付变化

需求改变时常发生，而良性依赖是那些不会“在实际中”造成危害的依赖关系，所以，良性依赖是相对的——需求改变可能使先前不易变的元素变得易变起来，从而良性依赖也变成了恶性依赖。

Robert C. Martin 在《敏捷软件开发》中提供的“只受一次愚弄”的策略很精辟。在我们最初编写代码时，假设变化不会发生；这时的设计很简洁，但对当时的需求，却是有效的、“不多不少的”。当变化发生时，我们就通过创建良性依赖，来隔离以后发生的同类变化；一般认为要通过创建抽象来隔离变化，而本章务实地认为只要是“不易变的”元素就可以。

## 21.3 案例：需求改变引起良性依赖变成恶性依赖

下面，通过几个应用实例，说明良性依赖原则如何和著名的“面向对象设计 5 大原则”结合，来“务实地应付变化”。

需求改变了，原先的良性依赖变成了恶性依赖，但我们“只受一次愚弄”。

比如，在开发一个需求跟踪工具的时候，起初可能仅需要支持保存为专有格式的“项目”文件，但后来又需要支持导出为 HTML 格式的网页。

让我按照敏捷软件开发过程，来讲述这个故事。

最开始的设计如图 21-1 所示，CReqMatrixDoc 调用 CProjectSaver 来保存自己。此时，所有需求就是支持“保存为专有格式的项目文件”，而且我们并没有预见到将来还需要以更多的形式保存，所以类 CProjectSaver 此时是“不易变的”，CReqMatrixDoc 对 CProjectSaver 的依赖是良性依赖，整个设计也是个“稳定的”设计。顺便说明，按照开放—封闭原则（Open—Closed Principle），这并不是一个好的设计；但按照当前的需求，这个设计却“不多不少”刚刚好，因为当前它是满足良性依赖原则的。

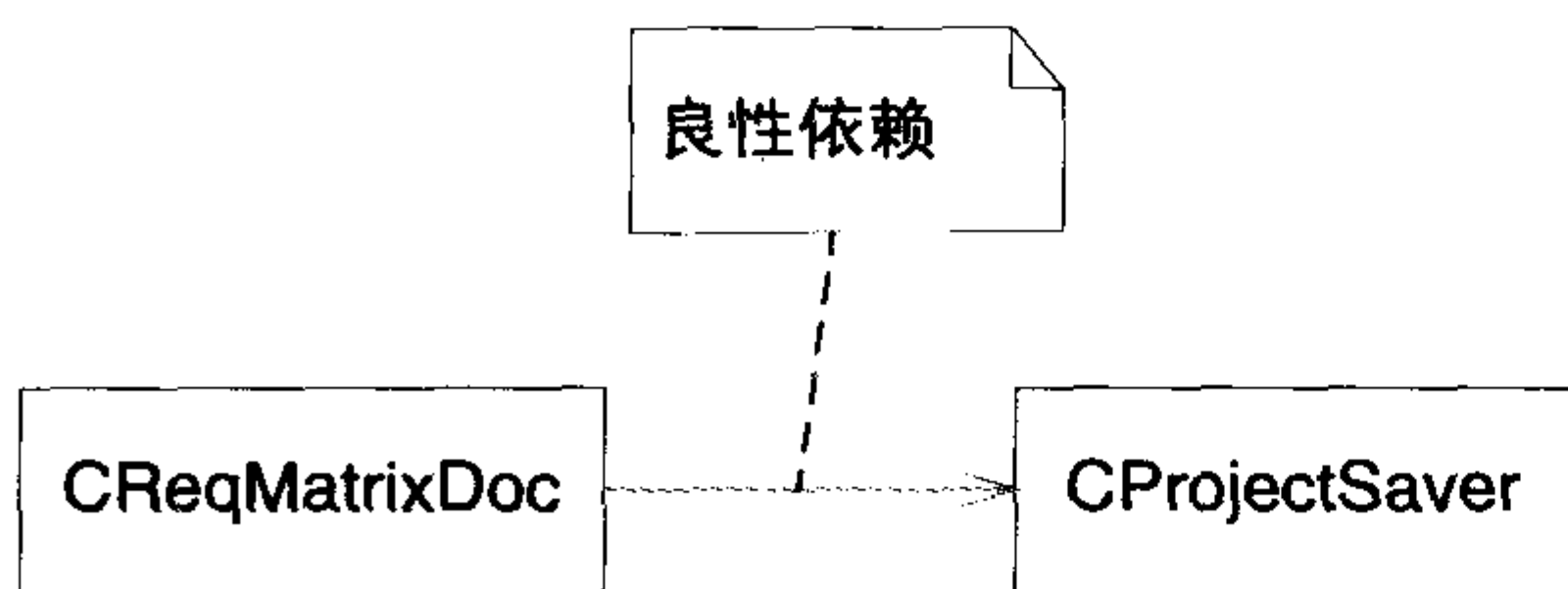


图 21-1 需求单一时，此直接引用为良性依赖

后来需求发生了变化，这个工具需要支持“导出为 HTML 格式的网页”的特性。是的，这个需求不管是客户新提出来的，还是设计人员在上一个迭代有意忽略了，总之在这个迭代周期需求发生了变化。于是，设计人员意识到，需求跟踪工具可能需要支持多种保存策略；如果不改变原来的设计，那么 CProjectSaver 就是“易变的”，因为它要支持可能不只一种新的保存策略。好了，如图 21-2 所示，设计虽然没有改变，但由于需求的改变，原来设计中的良性依赖，现在变成了恶性依赖，这意味着 CReqMatrixDoc 可能也要随着 CProjectSaver 的改变而改变，这不是一个灵活的设计。

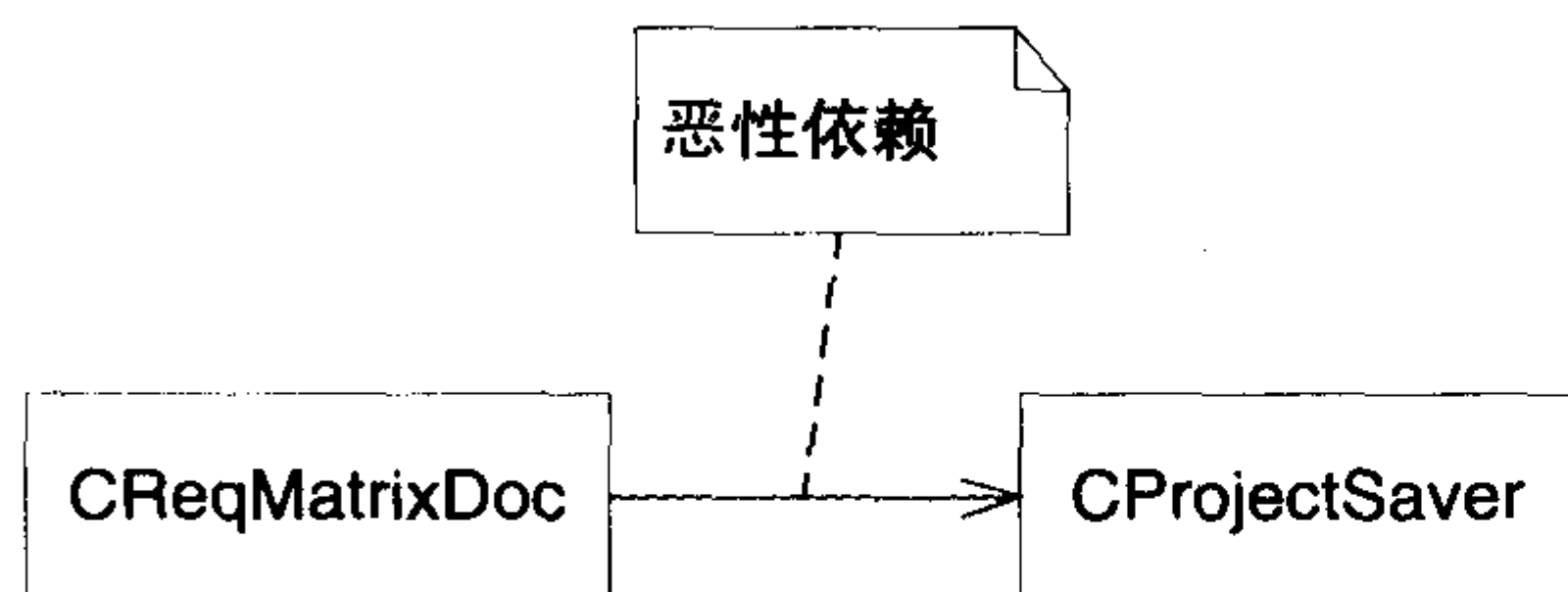


图 21-2 需求更多了，此直接引用成了恶性依赖

是的，代码出现了臭味（Smell），需要重构（Refactoring）。让我们谨遵 Martin Fowler 的教诲——不要将重构和添加新功能同时进行——这一步我们仅进行重构。我们要做的就是去除这个恶性依赖，采用依赖倒置原则（Dependency-Inversion Principle）惯用的“用两个抽象依赖代替一个具体依赖”策略，重构之后的设计如图 21-3 所示。我们引入了一个接口 CDocSaver，然后让 CProjectSaver 实现这个接口。一个设计良好的接口无疑是“不易变的”，所以不管是 CReqMatrixDoc 对 CDocSaver 的调用，还是 CProjectSaver 对 CDocSaver 的实现，都是良性依赖。重构完毕。

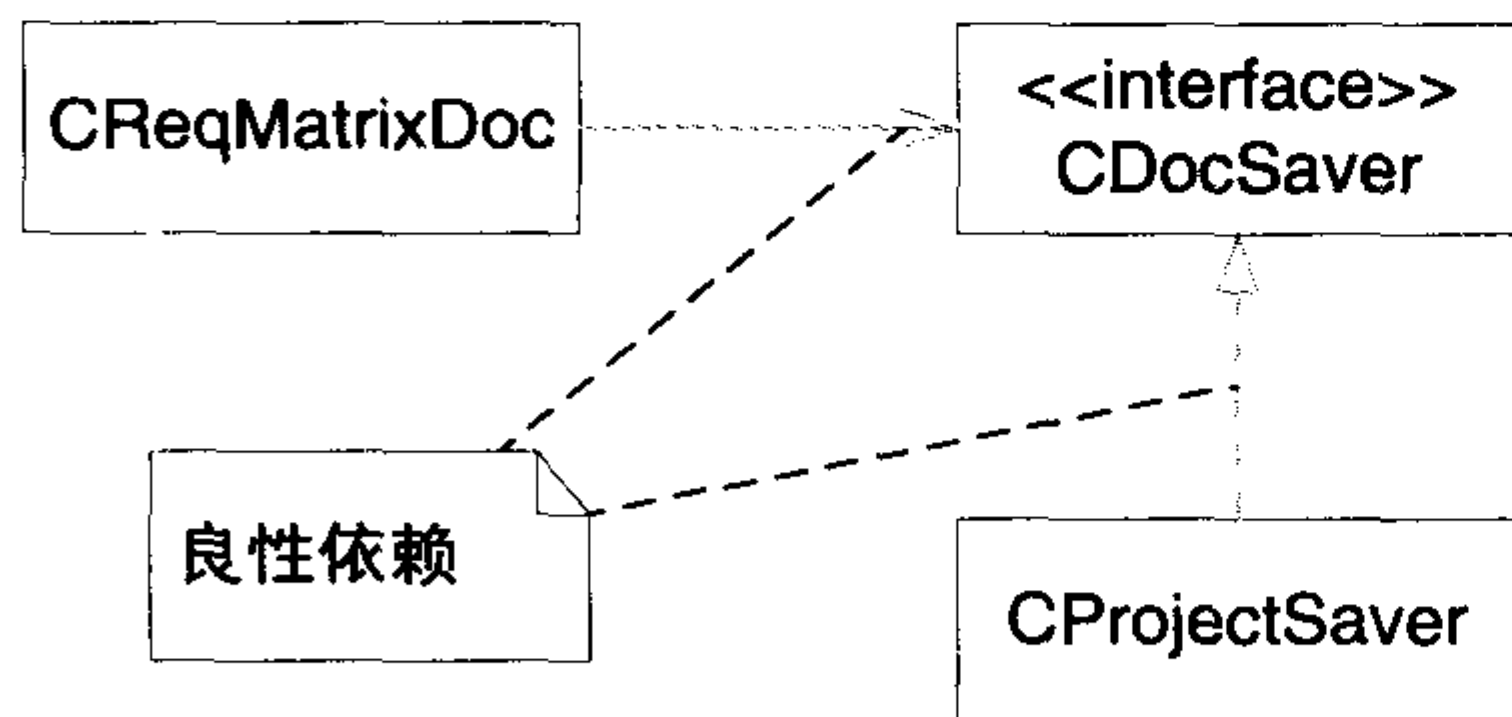


图 21-3 重构：引入接口

哈，新的设计非常易于扩充，我们只需新写一个 CHtmlSaver 来实现接口 CDocSaver，就离支持“导出为 HTML 格式的网页”不远了，如图 21-4 所示。咦？原来是策略模式。



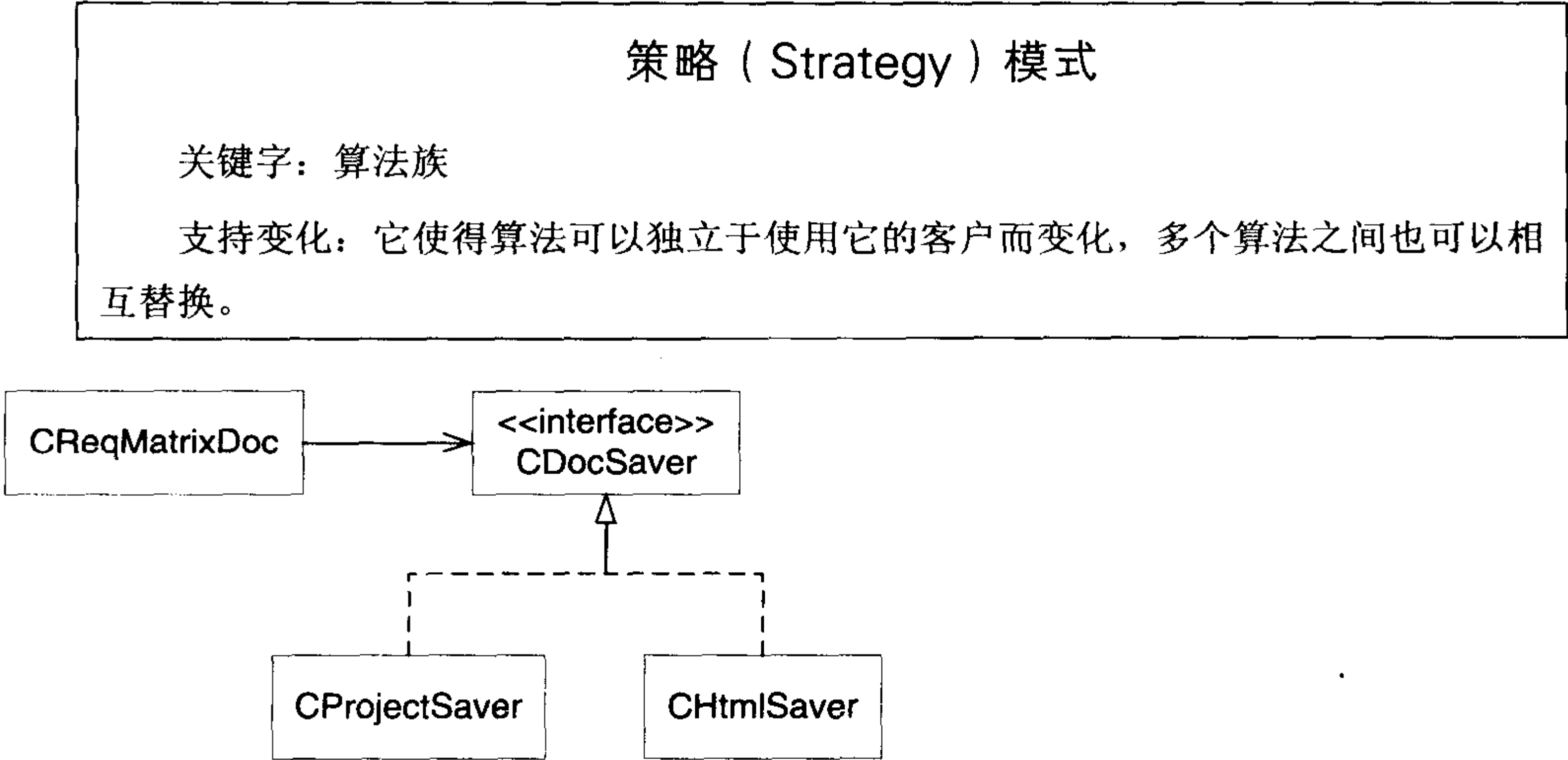


图 21-4 此时采用策略模式正是时候

## 21.4 案例：隔离第三方 SDK 可能造成的冲击

恶性依赖“作恶多端”。当恶性依赖中的被依赖元素变化时，依赖它的元素也可能要跟着变化；如果后者元素又在其他依赖关系中担当“被依赖元素”的角色，可能还会引起别的元素变化；这样，影响就会传播到很大的范围。

有时候，去除恶性依赖的代价比较大；还有时候，恶性依赖在所难免。我们应当如何？答案是，不去追求完美，而是务实地用良性依赖隔离恶性依赖造成的危害。

就让我来举个极端的例子——第三方 SDK。

我们要开发的是压缩工具，我们不可能漠视现有的第三方 SDK 的存在，它们的诱惑实在太大了。在第一个迭代周期，要支持 Zip 压缩格式，我们决定采用著名的 Info Zip 开发包。Info Zip 开发包并不在我们的控制之下——它的接口可能发生改变，当我们要使用更新的包时，我们可能面临不得不改动分散在很多类中的 Info Zip 使用代码的问题。所以我们要隔离这个我们控制不了的变化——对，就用 Adapter 模式——引入一个 CInfoZipAdapter 类来包装 Info Zip，如图 21-5 所示。这样，在 Info Zip 包升级时，我们仅需改动 CInfoZipAdapter 的实现就可以了。这个 CInfoZipAdapter 并不是一个抽象类，所以当前的设计并不满足依赖倒置原则（Dependency-Inversion Principle）推崇的“依赖于抽象”的要求；但 client 对 CInfoZipAdapter 的依赖关系是稳定的良性依赖，我们完全可以安心地远离过度设计（Over-engineering）。

适配器（Adapter）模式

关键字：已存在/不可预见 复用

支持变化：由于 Adapter 提供了一层间接，使得我们可以复用接口不符合我们需求的已存在的类，也可以使一个类（Adaptee）在发生不可预见的变化时，仅仅影响 Adapter 而不影响 Adapter 的客户类。

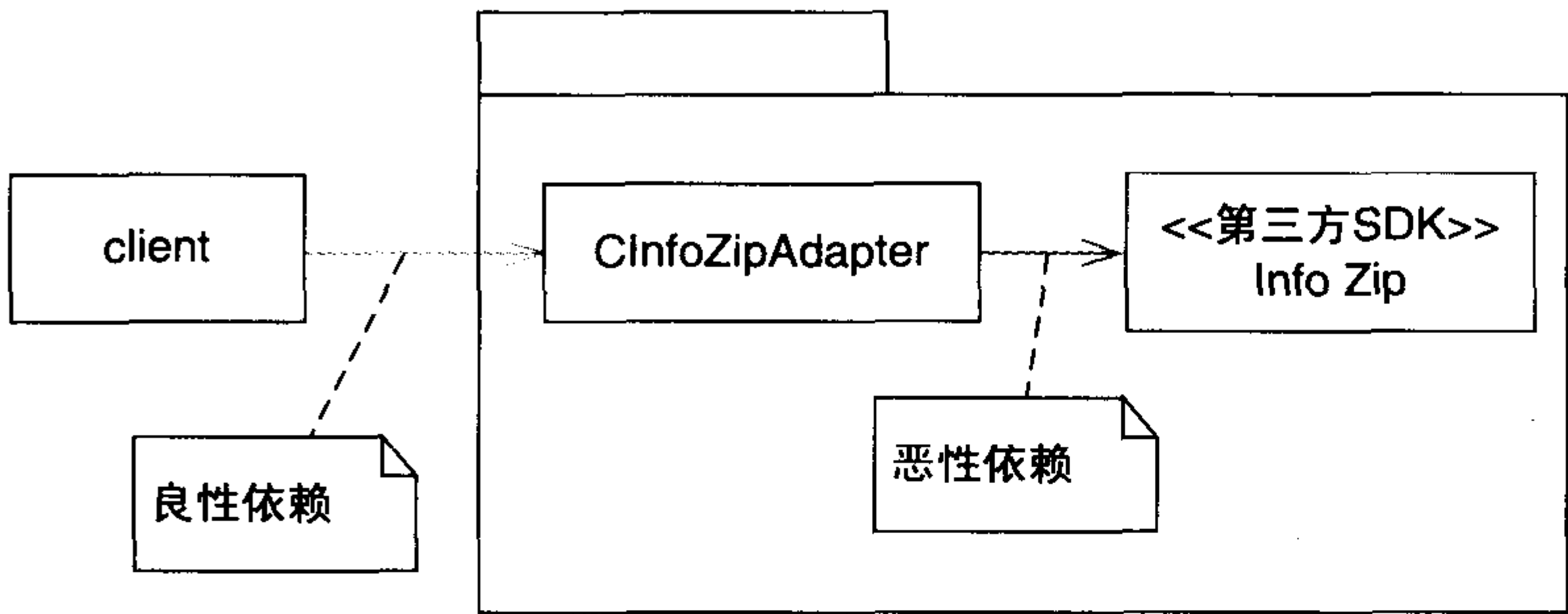


图 21-5 采用 Adapter 模式隔离第三方 SDK

谨遵敏捷宣言“经常性地交付可以工作的软件”的教诲，第一个迭代周期过后，我们发布了压缩工具的一个可以工作的版本。第二个迭代周期，我们需要使用另外一个第三方的开发包来支持新的压缩格式。显然，我们不应当让 CInfoZipAdapter 承担多于一个的职责，还是需要先重构。引入了接口 CCompressAlgo 供“外部”调用，如图 21-6 所示。

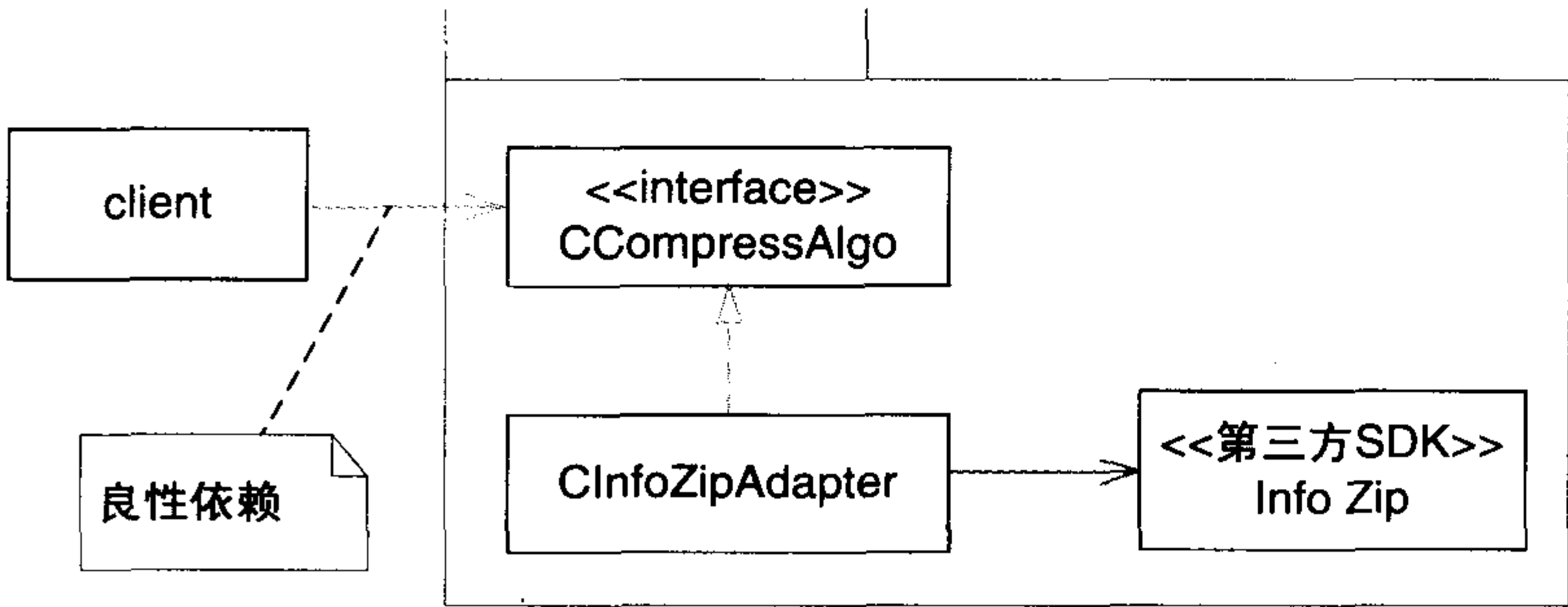


图 21-6 重构：引入 Adapter 接口

重构完毕，可以添加新功能了。从 CCompressAlgo “接口继承” 出来一个 COtherAdapter 来封装另一个第三方 SDK，如图 21-7 所示。哈，基本满意：（多个）client 对 CCompressAlgo 的良性依赖，使 client 的代码相当稳定；所有第三方 SDK 的不可控制的变化因素，都被妥善隔离。

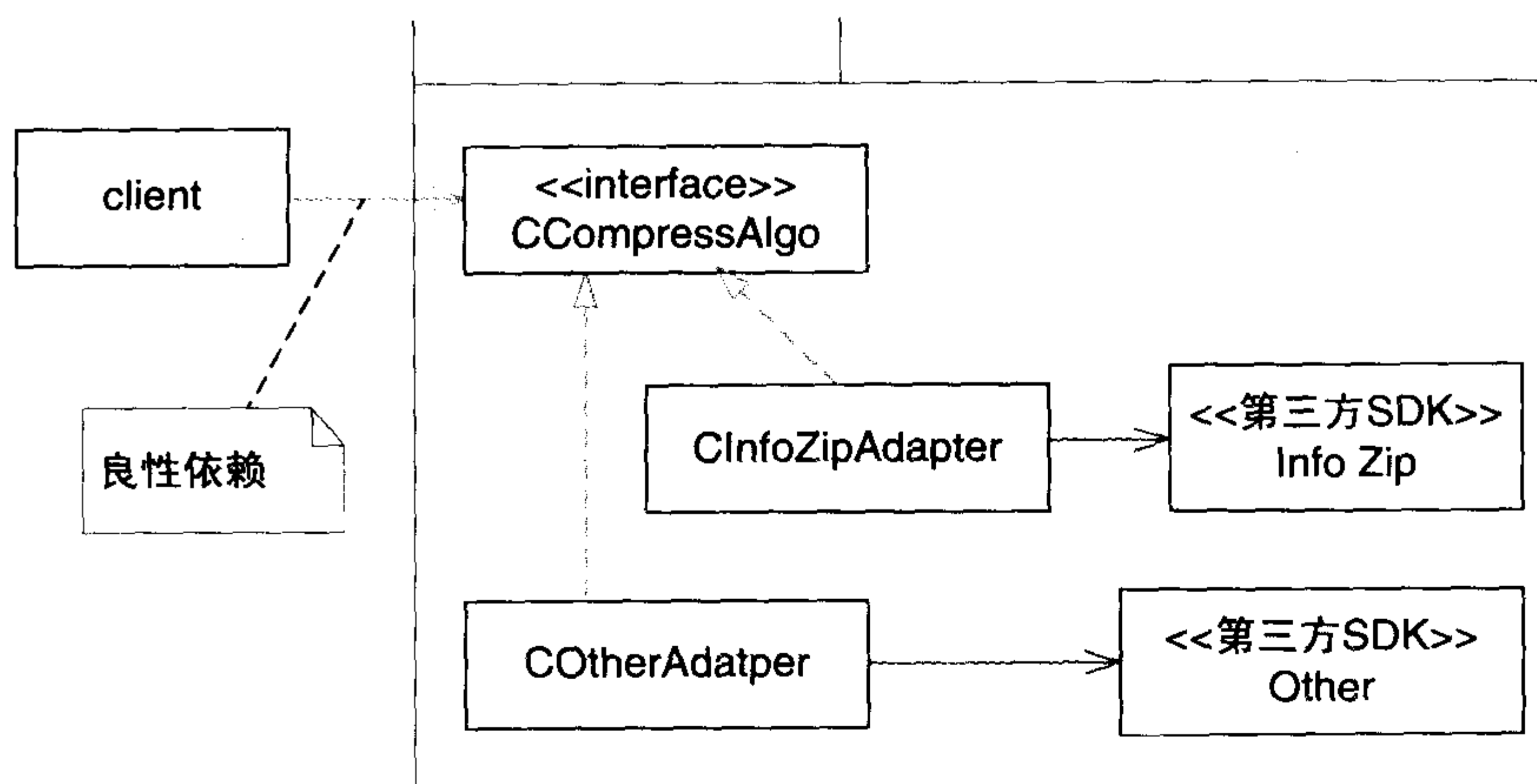


图 21-7 Adapter 接口可以有不同实现

## 21.5 案例：对具体类的良性依赖

良性依赖可以是对抽象基类的依赖，也可以是对具体类的依赖。其实，这种对具体类的良性依赖的例子是很多的，比如《设计模式》和《敏捷软件开发》中 Facade 模式的相关例子。下面，笔者举一个基于组件开发的例子，在“组件重用”这种“黑盒重用”日益盛行的今天，也许更具现实意义。

好的组件库，都恪守接口隔离原则（Interface-Segregation Principle），以达到“不应该强迫客户依赖于它们不用的方法”的目的。这其中包含了基于角色的设计（Role-based Design）的思想：协作被定义为“多个对象为了完成某种目标而进行的交互”；角色被定义为“特定协作中的对象的抽象”，它“仅定义了对对象特征的一个对某协作有意义的子集”；协作和角色的概念和现实世界很接近，我们很容易通过已有角色的组合来构造新的协作，以完成新的功能。

好了，看我们的需求：在多个 XML 文件中，查找那些包含特定名字的元素的文件。比如，我们可能希望知道哪些 XML 文件中包含名为 book 的元素，这在 XML 网页越来越多的今天，对搜索引擎无疑是很有用。我们使用微软的 DOM 组件来实现：CXmlFileSearcher 代表一个高层模块，它的职责就是上面描述的需求；而具体的打开 XML 文件的工作，它交给 IXMLDOMDocument 来做；具体的读取 XML 文件中的每个元素的名字的工作，交给 IXMLDOMNode 来做；它是个典型的 Facade 模式，如图 21-8 所示。

## 外观（Facade）模式

关键字：子系统 高层接口

支持变化：它实现了子系统与客户之间的松耦合关系，而子系统内部的功能部件往往是紧耦合的。松耦合关系使得子系统的组件变化不会影响到它的客户。

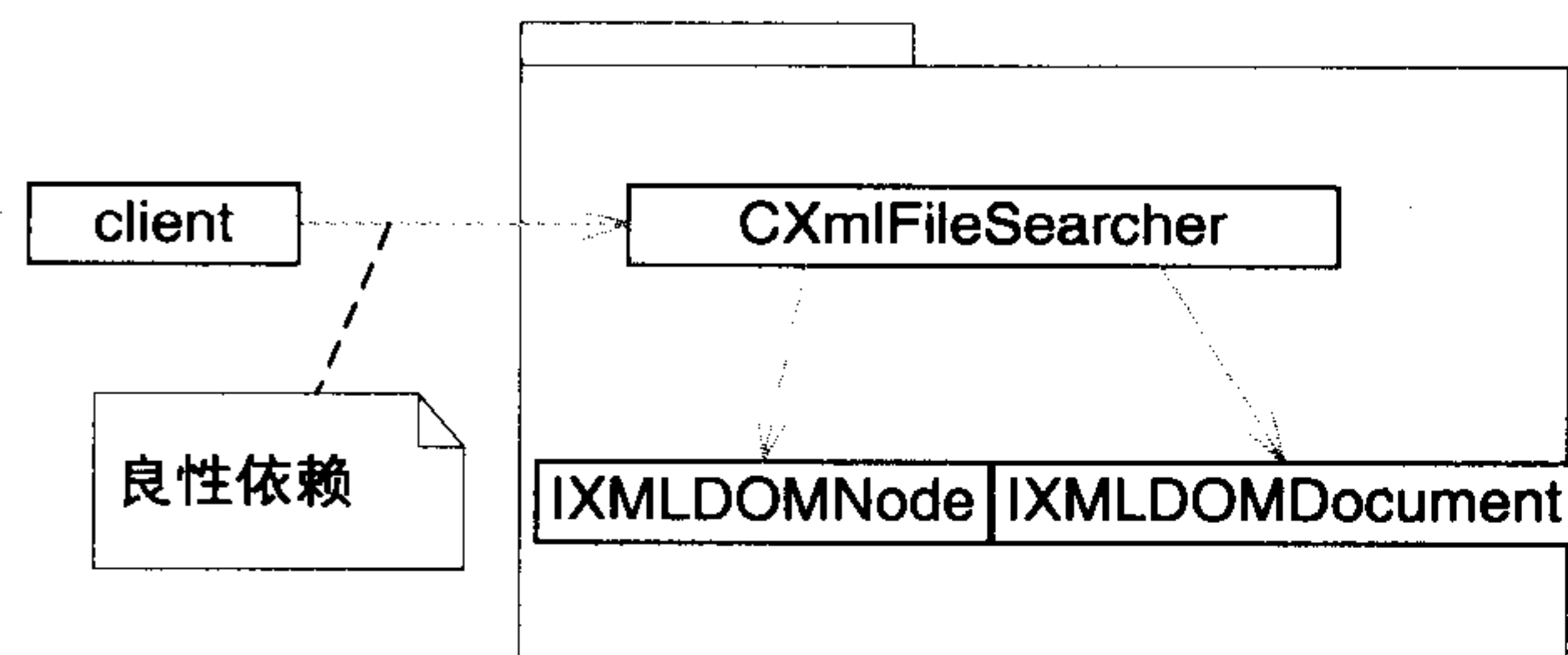


图 21-8 基于 Façade 模式的设计

值得说明的是，按照依赖倒置原则（Dependency-Inversion Principle），应当做到“高层模块不应该依赖于低层模块，二者都应该依赖于抽象”。就是说，我们应当为 CXmlFileSearcher 提供一个抽象基类，供外部的高层模块 client 调用，这在当前的需求之下，未免有过度设计（Over-engineering）之嫌。这也正好体现了良性依赖原则的“务实”优点。

当然，我们并不是一味地回避使用抽象基类的“抽象耦合”。这不，没过多久，新需求出现了，不仅要搜索元素名，还要搜索属性名和文本格式的内容。还是首先仅做重构，引入抽象基类 CSearcher，原先在 CXmlFileSearcher 中实现的搜索策略移到具体类 CEleNameSearcher 中，如图 21-9 所示。需要特别说明的是，现在引入抽象和上一步引入抽象，是完全不同的两回事：现在引入抽象是基于实实在在的需求，而上一步引入抽象是基于猜测，其接口很可能不能满足刚刚提出来的新需求。

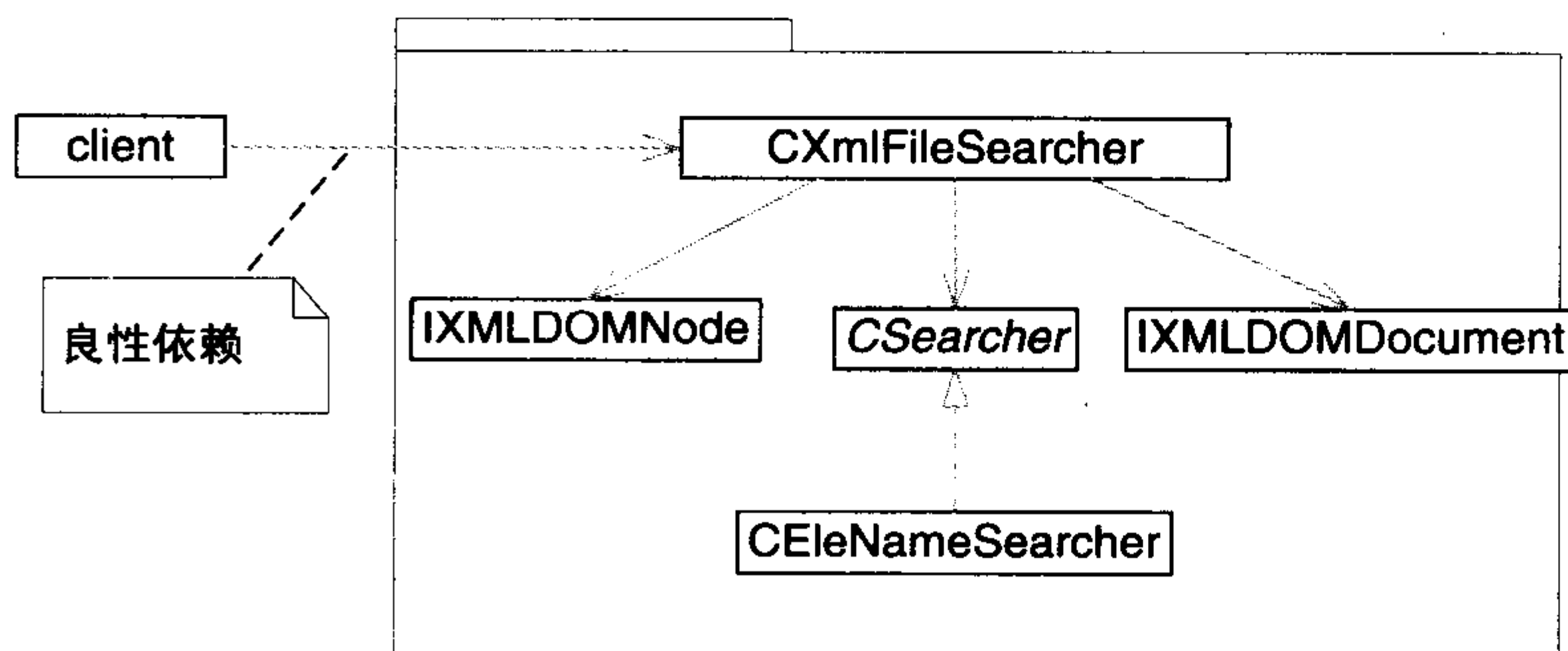


图 21-9 引入抽象基类 CSearcher



重构完毕，可以增加新功能了。运用策略模式，将搜索属性名和文本格式的内容分别实现作具体类 CAttrSearcher 和 CTextSearcher，如图 21-10 所示。

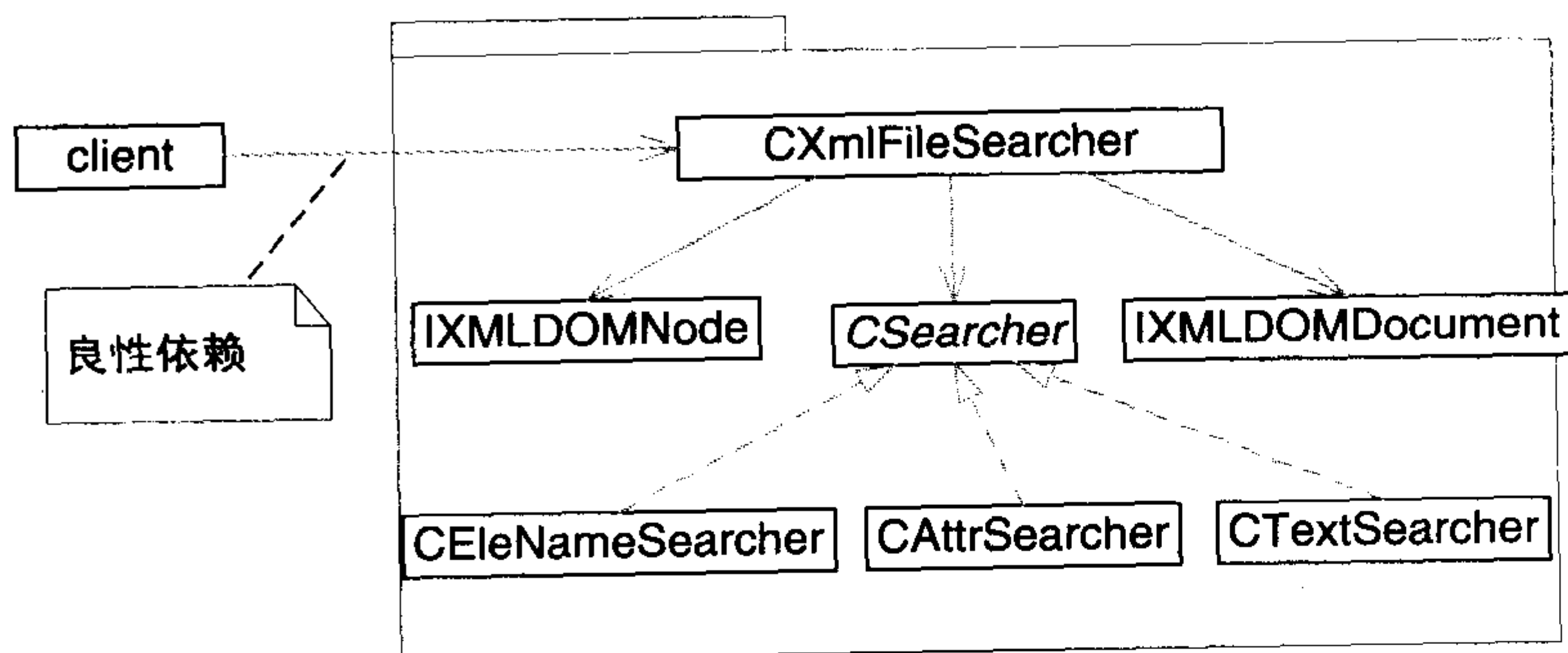


图 21-10 进一步引入策略模式

## 21.6 总结：如何处理好依赖关系

“依赖”是和“变化”紧密联系在一起的概念。由于依赖关系的存在，变化在某处发生时，影响会波及开去，造成很多修改工作，这就是依赖的危害。可以说，变化是始作俑者，依赖是助纣为虐。

我们可以不去拥抱变化吗？不可以。未来将越来越不可预测，这是新经济最具挑战性的方面之一。商务和技术上的瞬息万变会产生变化，这既可以看作要防范的威胁，也可以看作应该欢迎的机遇。

既然变化不可避免，我们所能做的就是处理好依赖关系，将变化造成的影响的波及范围尽量减小。

下面总结一下“面向对象设计 5 大原则”和良性依赖原则在应付变化方面的作用。

**单一职责原则 (Single-Responsibility Principle)。**“对一个类而言，应该仅有一个引起它变化的原因”。本原则是我们非常熟悉地“高内聚性原则”的引申，但是通过将“职责”极具创意地定义为“变化的原因”，使得本原则极具可操作性，尽显大师风范。同时，本原则还揭示了内聚性和耦合性是“一物两面”的关系，为了降低耦合性，基本途径就是提高内聚性；如果一个类承担的职责过多，那么这些职责就会相互依赖，一个职责的变化可能会影响另一个职责的履行。其实 OOD 的实质，就是合理地进行类的职责分配。

**开放封闭原则 (Open-Closed Principle)。**“软件实体应该是可以扩展的，但是不可修改”。本原则紧紧围绕变化展开，变化来临时，如果不必改动软件实体的源代码，就能扩充它的行为，那么这个软件实体的设计就是满足开放封闭原则的。如果我们预测到某种变化，或者某种变化发生

了，我们应当创建抽象来隔离以后发生的同类变化。在 Java 中，这种抽象指抽象基类或接口；在 C++ 中，这种抽象是指抽象基类或纯抽象基类。当然，没有对所有情况都贴切的模型，我们必须对软件实体应该面对的变化做出选择。

**Liskov 替换原则**（Liskov-Substitution Principle）。“子类型必须能够替换掉它们的基类型”。本原则和开放封闭原则关系密切，正是子类型的可替换性，才使得使用基类型的模块无需修改就可扩充。Liskov 替换原则从基于契约的设计演化而来，契约通过为每个方法声明“先验条件”和“后验条件”；定义子类时，必须遵守这些“先验条件”和“后验条件”。当前，基于契约的设计发展势头正劲，对实现“软件工厂”的“组装生产”梦想是一个有力的支持。

**依赖倒置原则**（Dependency-Inversion Principle）。“抽象不应依赖于细节，细节应该依赖于抽象”。本原则几乎就是软件设计的正本清源之道。因为人解决问题的思考过程是先抽象后具体，从笼统到细节的，所以我们先生产出的势必是抽象程度比较高的实体，而后才是更加细节化的实体。于是，“细节依赖于抽象”就意味着后来的依赖于先前的，这是自然而然的重用之道。而且，抽象的实体代表着笼而统之的认识，人们总是比较容易正确认识它们，而且它们本身也是不易变的，依赖于它们是安全的。依赖倒置原则适应了人类认识过程的规律，是面向对象设计的标志所在。

**接口隔离原则**（Interface-Segregation Principle）。“多个专用接口优于一个单一的通用接口”。本原则是单一职责原则用于接口设计的自然结果。一个接口应该保证，实现该接口的实例对象可以只呈现为单一的角色；这样，当某个客户程序的要求发生变化，而迫使接口发生改变时，影响到其他客户程序的可能性最小。

**良性依赖原则**。“不会在实际中造成危害的依赖关系，都是良性依赖”。通过分析不难发现，本原则的核心思想是“务实”，很好地揭示了极限编程（Extreme Programming）中“简单设计”和“重构”的理论基础。本原则可以帮助我们抵御“面向对象设计 5 大原则”以及设计模式的诱惑，以免陷入过度设计（Over-engineering）的尴尬境地，带来不必要的复杂性。

## 第 22 章 基于角色的设计：从理论到实践

---

面向对象设计最困难的部分是将系统分解成对象集合。因为要考虑许多因素：封装、粒度、依赖关系、灵活性、性能、演化、复用等等，它们都影响着系统的分解，并且这些因素通常还是相互冲突的。

——GOF,《设计模式》

《射雕英雄传》中，生性好玩的老顽童周伯通，被困桃花岛十五年，长年枯坐，十分无聊，于是想出了左手和右手打架的法子来取乐，于是悟出了分心二用、双手互搏的奇妙武功。常言道，“一心不能二用”，而这双手互搏之术，却正是要人一心二用，从“左手画方，右手画圆”开始练习，练成后能双手同时使出不同武功来攻击敌人。

这种“双手同时使出不同武功”的功夫，使我不禁想到了软件开发。最简单的，当只有一个人编写软件时，你要一会儿编写这个类，一会儿又编写那个类；你可能既要编写业务层模块，又要编写表现层模块，而这需要不同的领域知识；甚至你在编写一个模块的时候，也会想到这个模块的客户如何调用它……

其实，稍做分析，我们就能得出结论：即使是只有一个软件人员单兵作战，也存在一个“协作”的问题——只不过是自己和自己协作罢了。究其原因，是因为人类解决复杂问题的基本手段就是分而治之。为了解决一个大的问题，可以：（1）把它分成两个或多个更小的问题；（2）分别解决每个小问题；（3）把各小问题的解答组合起来，即可得到原问题的解答。

软件工程使软件开发从手工作坊上升到团队开发模式。如果说，对于单兵作战的个人开发而言，分而治之的架构设计仅仅是“善待自己”的话，那么，对集团作战的团队开发而言，合理的分而治之的松耦合高聚合架构，就是不可或缺的“求生之道”。

本章结合实际案例，阐述基于角色的设计（Role-based Design）如何通过角色的抽象与协作，来隔离不同级别的程序单元，使它们能够相对独立地变化，进而使团队开发成员职责明晰，促进团队开发高效进行。

## 22.1 基于角色的设计理论

用面向对象的观点去观察世界，我们可以发现许多不同种类的对象，它们都有自己的内部状态和运动规律；而不同对象之间的相互联系和相互作用，就构成了完整的客观世界。面向对象的开发方法论以对象为基本着眼点，以模拟为手段，建立问题空间的解空间；相应地，用面向对象方法开发出来的软件系统，由一系列相互协作的对象组成，每个对象在不同协作中承担特定角色。

基于角色的设计，其要旨在于使用角色组装协作。协作被定义为“多个对象为了完成某种目标而进行的交互”。角色被定义为“特定协作中的对象的抽象”，它“仅定义了对象特征的一个对某协作有意义的子集”。

协作和角色的概念，同时存在于软件设计和现实世界中，图 22-1 展示了“老张”扮演的三个角色——父亲、丈夫、老师。

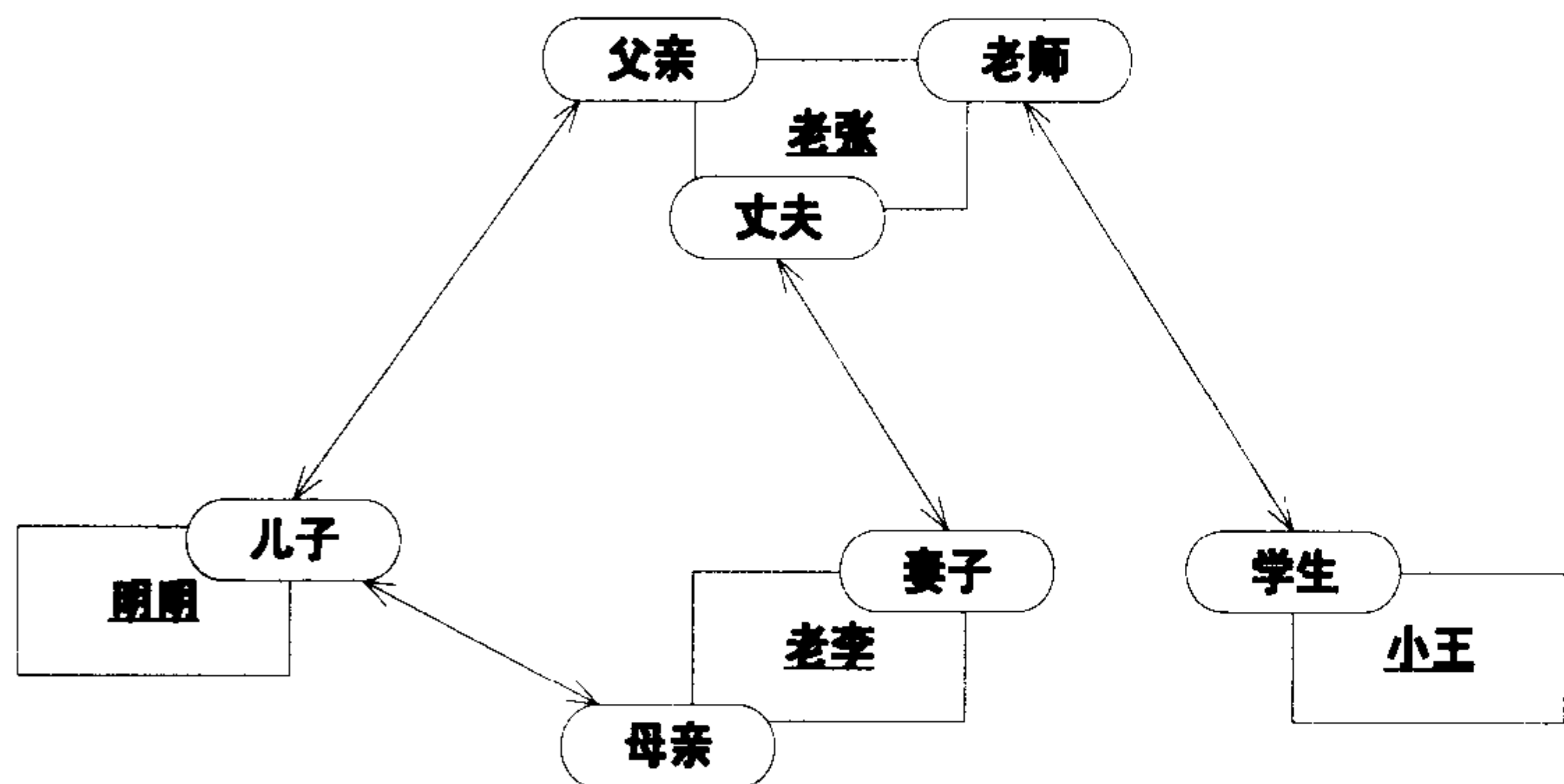


图 22-1 现实中的协作与角色

基于角色的设计的意义在于：这种设计带来了良好的可重用性，节省了开发资源；通过已有角色的组合，就可以构造新的协作来完成新的功能，这种方式非常自然和容易理解；至于非功能性的好处，这种设计还具有松耦合、高聚合的优点。

## 22.2 基于角色的设计与团队开发

随着软件规模的扩大，软件开发早已进入团队开发的时代；尤其是在科技和经济飞速发展的现代社会，很多软件项目本身就要求在一定的时间内完成，否则就失去了意义。团队开发的成功与否，受到多方面因素的制约，而基于角色的设计带来的良好架构，为团队开发打下了良好基础，其奥秘全在“协作”二字。



团队开发就是协作开发，每个团队成员，在协作开发中承担一定的职责；对开发人员而言，每人会负责一定数量的类的开发和维护。团队成员之间的协作，和其承担的类之间的协作，并非没有关系；恰恰相反，软件系统中类的依赖关系是开发团队中人的依赖关系的根源。一图胜千言，图 22-2 说明了这一点。

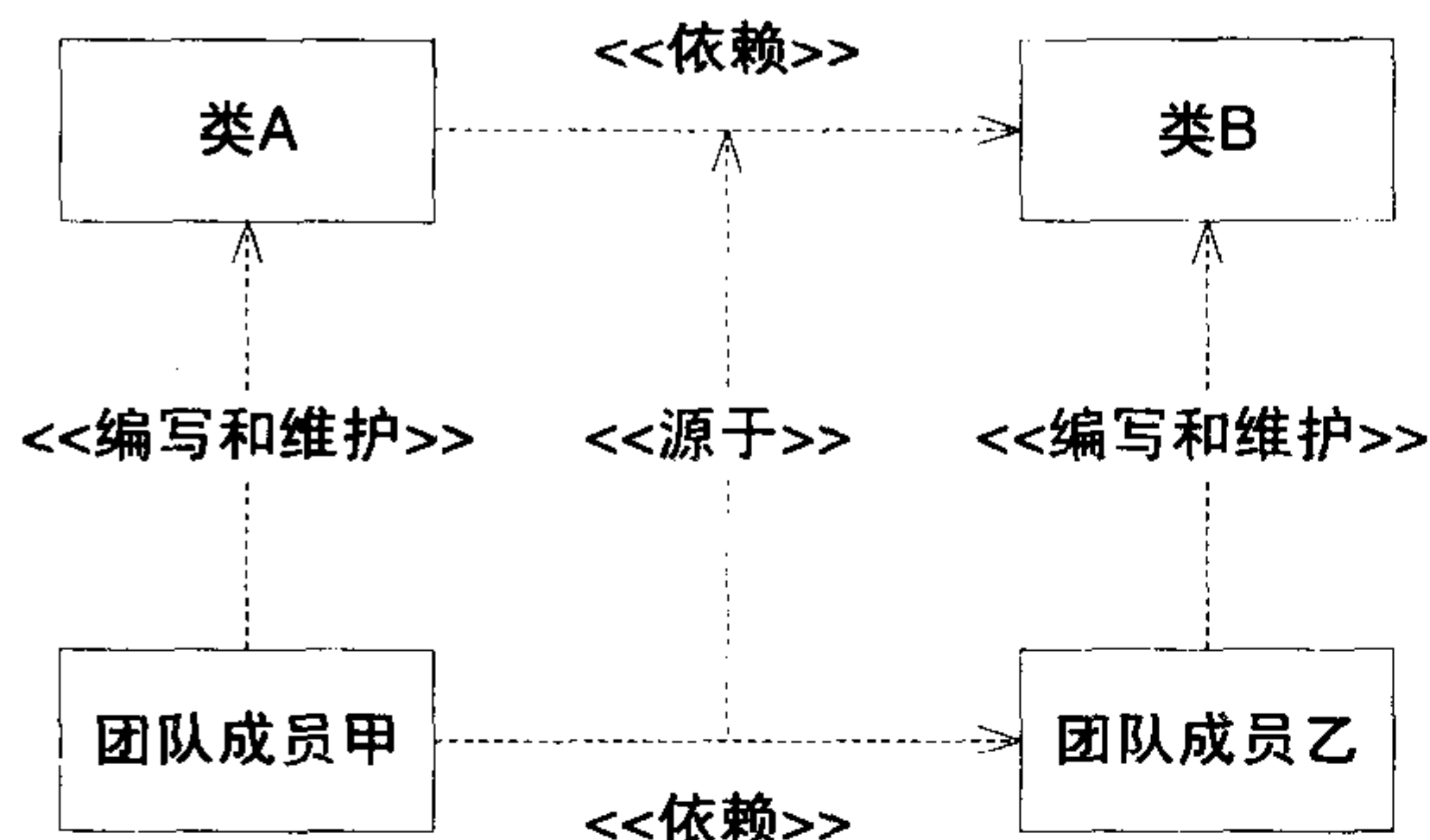


图 22-2 类的依赖与人的依赖

在实践中，CRC 技术对这一点的体现最为充分和直接。著名的 CRC（类—职责—协作）技术，使用卡片和角色扮演的方法，帮助人们将系统设计为一系列协作的对象。具体讲，CRC 技术以索引卡片代表每个类，设计组的成员扮演类的角色，以对话的形式模拟系统的行为。卡片使得对象的思想更加具体，而角色扮演则促进了对对象协作的认识，两者相得益彰，相互促进，意义可谓大矣。

本章无意探讨开发组织的组织结构问题，但笔者认为团队开发的关键所在，乃是处理好开发成员之间的依赖关系——我们应当尽量减少开发成员之间的依赖关系。既然软件系统中类的依赖关系是开发团队中人的依赖关系的根源，我们就应当为软件设计合理的架构，处理好类之间的依赖关系。而基于角色的设计，既能够将一个软件系统划分成几个相对独立的子系统，也能够将一个子系统划分成几个相对独立的模块；这样，就理顺了负责开发不同子系统的成员之间，及同一子系统中负责不同模块的成员之间的依赖关系，使团队开发井井有条，高效有序。

## 22.3 基于角色的设计实践

没有类的协作，就没有面向对象的软件设计。作为一种面向对象设计方法论，基于角色的设计强调的是，系统、子系统、模块、类等这些不同级别的抽象单元之间的协作，不应当直接进行，而应当转而通过“抽象角色的协作”来间接完成。

首先，什么叫通过“抽象角色的协作”来间接完成抽象单元之间的协作呢？

简单而言，就是先为协作建立抽象模型，再分别具体实现协作中的角色。比如唐伯虎卖画，

现画现卖，买者如云，队伍排得老长；按照基于角色设计的方法，就可以建立“生产者—消费者—排队”的抽象模型。

“自由来自顺应规律”。基于角色的设计之所以强大，是因为它符合面向对象的根本原则。基于角色的设计符合依赖倒置原则（Dependency-Inversion Principle）。依赖倒置原则规定，抽象不应依赖于细节，细节应该依赖于抽象。本原则几乎就是软件设计的正本清源之道。因为人解决问题的思考过程是先抽象后具体，从笼统到细节的，所以我们先生产出势必是抽象程度比较高的实体，而后才是更加细节化的实体。于是，“细节依赖于抽象”就意味着后来的依赖于先前的，这是自然而然的重用之道。而且，抽象的实体代表着笼而统之的认识，人们总是比较容易正确认识它们，而且它们本身也是不易变的，依赖于它们是安全的。

依赖倒置原则是面向对象设计的标志所在，它适应了人类认识过程的规律。而软件开发就是这样一个认识问题解决问题的过程，基于角色的设计强调“先为协作建立抽象模型”，符合人类认识过程的规律，自然也理顺了团队的协作开发过程。

进一步地，来谈谈角色。

并不是随随便便哪个类都可以称为角色的，基于角色的设计对角色有严格的定义。角色是“特定协作中的对象的抽象”，它“仅定义了对对象特征的一个对某协作有意义的子集”。根据这个定义，我们很自然想到了著名的接口分离原则（Interface Separation Principle），该原则信奉“多个专用接口优于一个单一的通用接口”的思想，因为“任何接口都应当具有高内聚性”，以便“保证实现该接口的类的实例对象可以只呈现为单一的角色”。看来，基于角色的设计还真蕴藏了不少学问，不可小觑。

最后，从 UML 类图，可以很自然地导出基于角色的设计方案，这在 UML 已经成为建模语言标准的今天，实在是一条值得重书一笔的优点。举例说明，图 22-3 是一个再简单不过的类图。

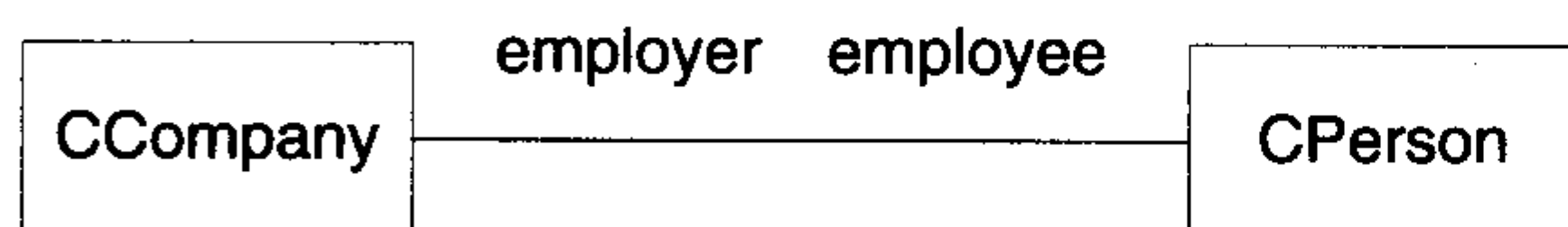


图 22-3 关联两端标明了角色名的类图

我们可以很自然地导出图 22-4 所示的设计。

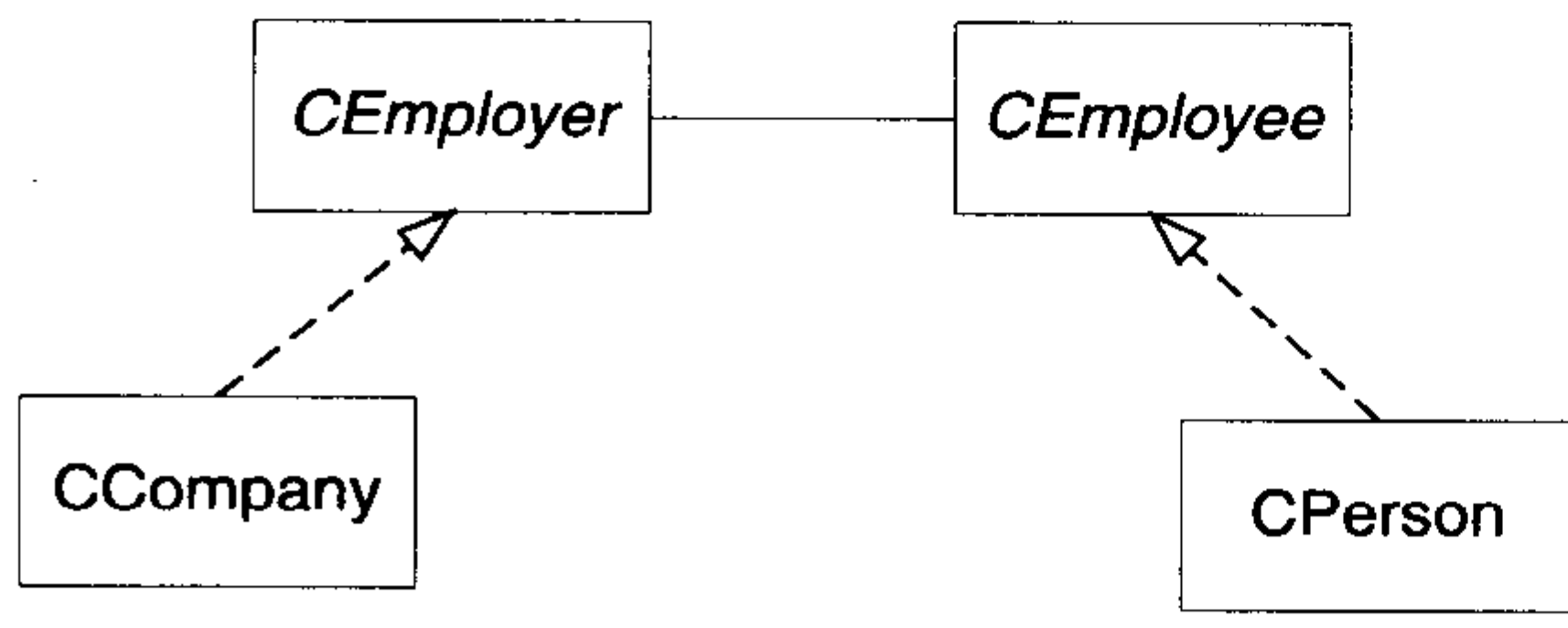


图 22-4 角色名已转变成接口的类图

## 22.4 基于角色的设计案例

下面，通过具体案例，说明基于角色的设计是如何通过角色的抽象与协作，来隔离不同子系统以及同一子系统的不同模块，使这些子系统和模块，能够在相互协作的大背景下尽可能地相对独立；于是，开发团队成员之间，也能够在相互协作的大背景下，可以尽可能地相对独立。

### 22.4.1 项目简介

这是一个网管软件，有多个子系统组成。比如，拓扑显示子系统负责显示被管理网络的拓扑图，设备发现子系统负责发现网络中的可管理设备，设备轮询子系统负责监控网络设备的状态。

### 22.4.2 通过基于角色的设计组织子系统之间的协作

如图 22-5 所示。首先，Topo 子系统扮演了 PollerWaiter 的角色，该角色负责处理 Poller 子系统产生的 StatusChanged 消息。其次，Topo 子系统还扮演 DiscoverWaiter 的角色，该角色负责处理 Discoverer 子系统产生的 NodeDiscovered 消息。还有，Discoverer 扮演 Discoverer 的角色，该角色负责处理 Topo 子系统产生的 DiscoverCommand 命令。

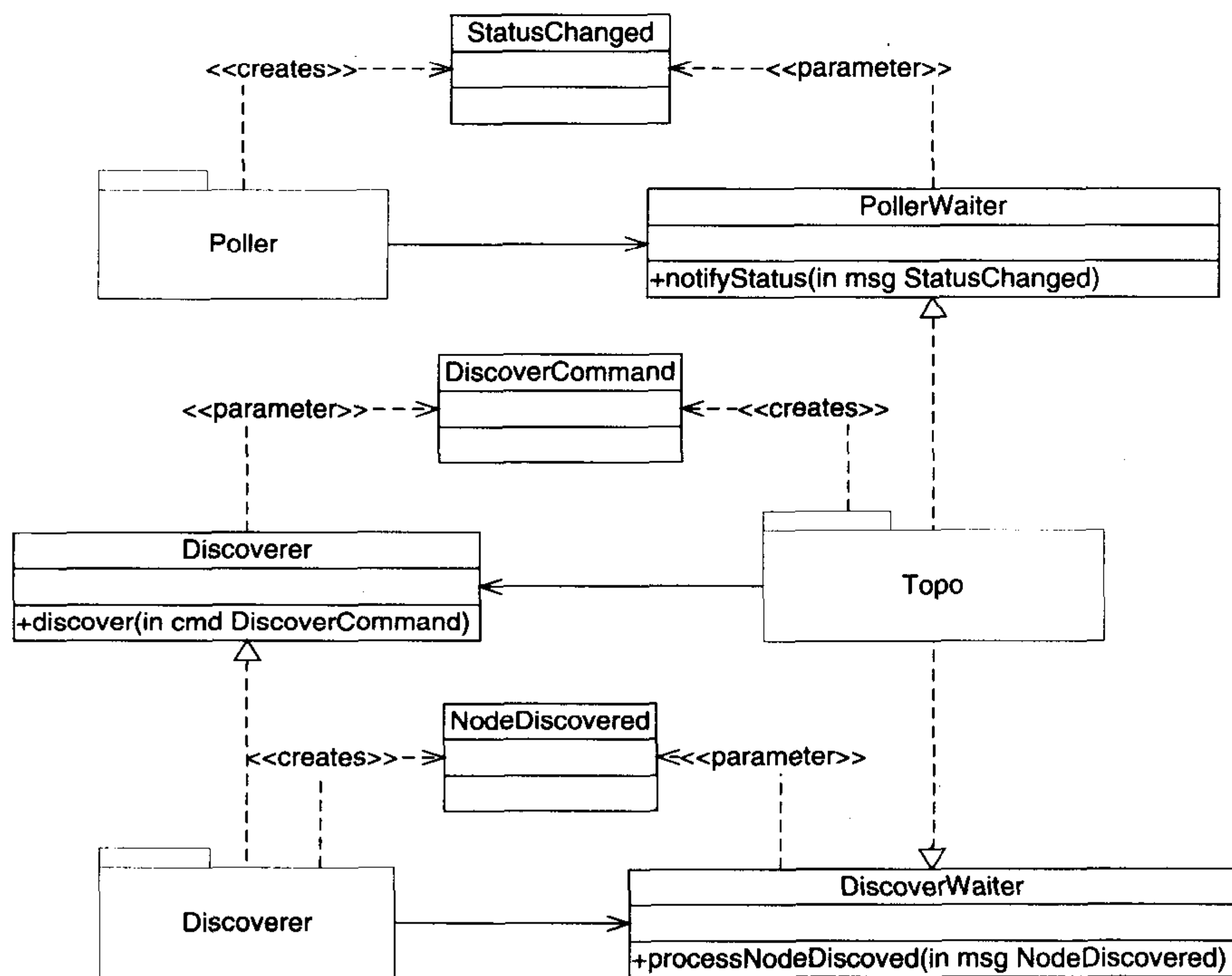


图 22-5 基于角色的设计用于组织子系统之间的协作

上述基于角色的设计方案，可以良好地隔离子系统，使不同子系统独立地变化，使不同开发小组间尽可能独立。比如，Discoverer 子系统负责发现网络中的可管理设备，但它不需要网络拓扑图的显示，它将发现信息抽象成 NodeDiscovered 消息，通过 DiscoverWaiter 接口通知给 Topo 子系统。Topo 子系统负责解析事先定义好的 NodeDiscovered 消息，来“制导”拓扑图的显示逻辑。这样，Discoverer 子系统和 Topo 子系统之间的交互，通过 DiscoverWaiter 接口和 NodeDiscovered 消息组成的模型来抽象。该模型抽象程度很高，是对系统的最笼而统之认识，稳定程度非常高。该模型为 Discoverer 子系统和 Topo 子系统的交互建立了稳定的契约，负责 Discoverer 子系统和 Topo 子系统的开发人员，可以相当自由地独立开发了。

而且，该抽象模型有极大的灵活性，它支持扩充。如图 22-6 所示，由 Discoverer 子系统产生的，由 Topo 子系统负责处理的 NodeDiscovered 消息可以方便地扩充。比如，EquipmentsDiscovered 消息可以用来精确地通知每一台设备的发现，有利于提高用户界面的响应速度；而 SubnetDiscovered 消息只有在整个子网都发现完毕的时候才会被发出，优点是通信开销小，缺点是用户界面的响应时间较长。

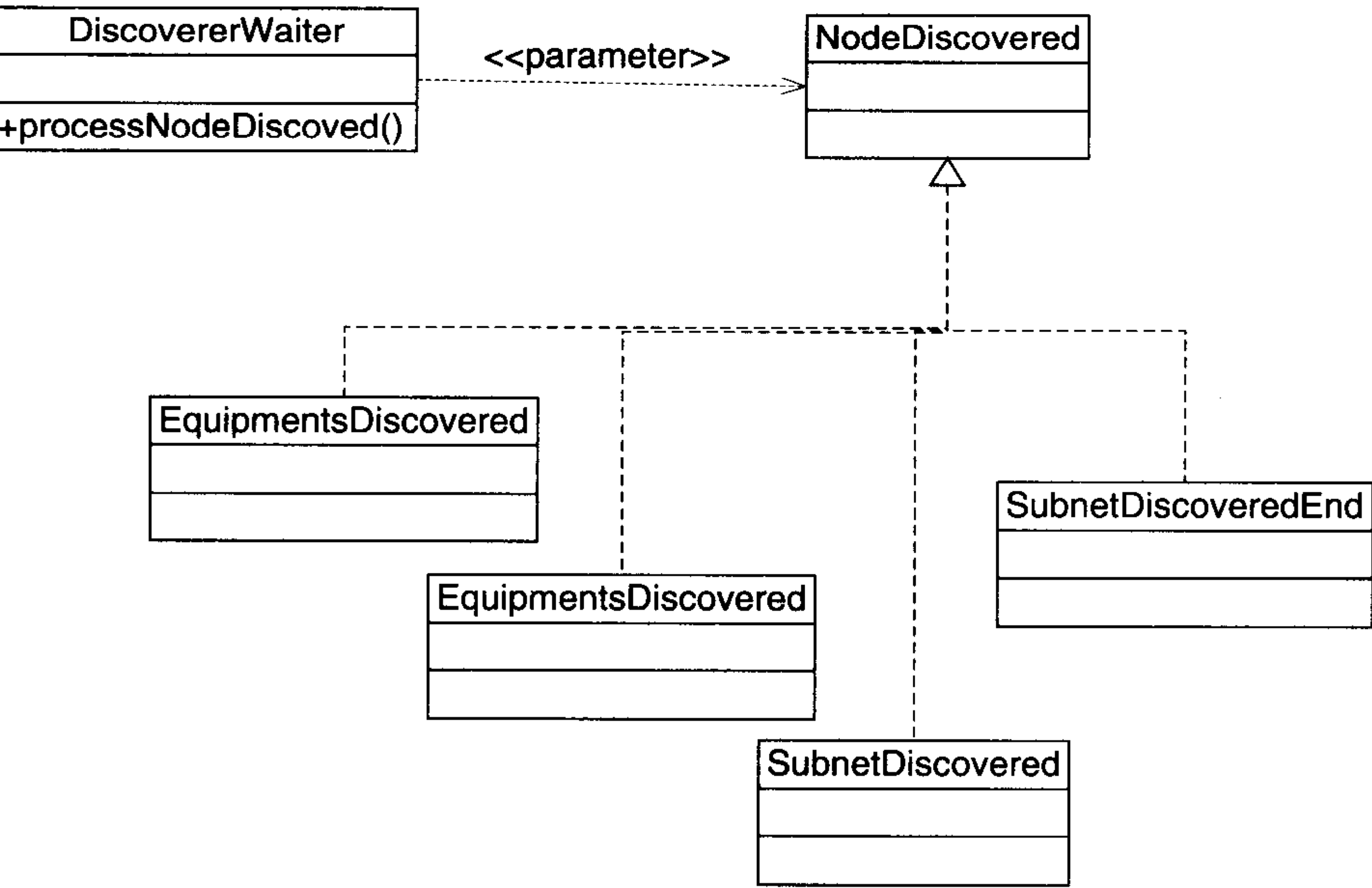


图 22-6 NodeDiscovered 消息类层次

### 22.4.3 通过基于角色的设计组织同一子系统内不同模块之间的协作

如图 22-7 所示，首先是 GeneralModel 扮演了被观察者的角色，而 TopoGraphModel 扮演了观察者的角色。这个著名的设计模式完美地抽象了一对多的通知机制，而且支持观察被观察的关系动态改变。



其次，TopoGraphModel 还扮演了 StatusChangedRender、GeneralModelChangedRender 和 ShowCustomizedHandler 的重角色。这体现了接口分离原则，其背后的秘密在于，要为不同的预期协作提供不同的角色。比如 ShowCustomizedHandler 是为用户定制拓扑图显示方式提供的，它只改变 TopoGraphModel，而不会影响业务级的模型 GeneralModel。

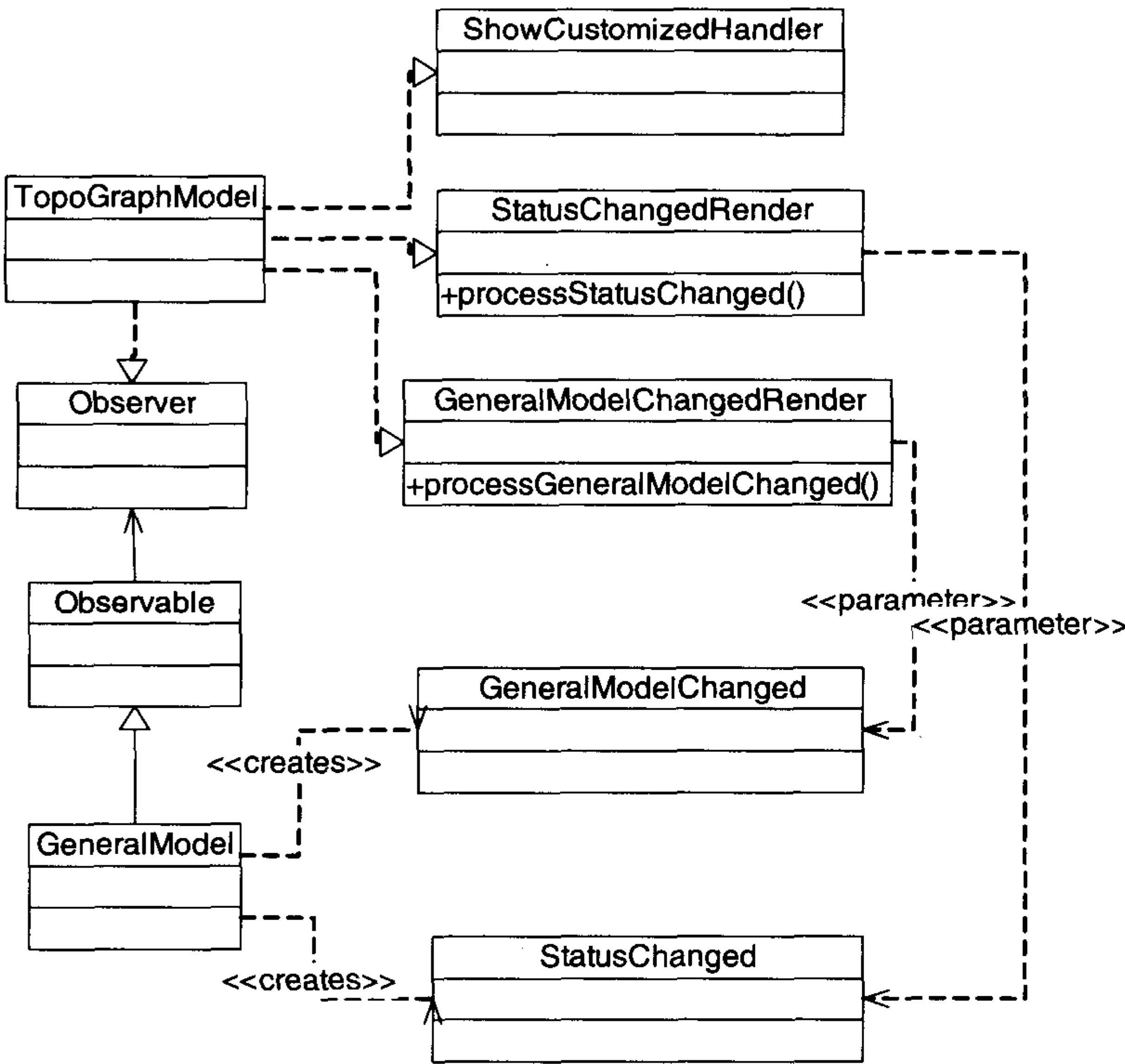


图 22-7 基于角色的设计用于组织子系统内协作

上述基于角色的设计方案，能够良好地为模块解耦，使同一子系统的不同模块独立地变化，使同一开发小组的不同成员尽可能独立工作。比如，对于拓扑图显示，有很多现成的第三方框架可以用，而上述设计将 Topo 子系统地划分成了多个相对独立的模块——TopoGraphModel 和相关类完全将第三方框架和 Topo 子系统其他类隔离，负责 TopoGraphModel 的工程师完全可以决定框架的选择和更换，而不会对负责 GeneralModel 等类的工程师造成冲击。

## 22.5 基于角色的设计与面向对象分析

没有银弹，基于角色的设计当然也不是！本章虽然主要讨论面向对象设计方面的问题，但在本章的最后想要提醒大家的是，基于角色的设计是否能带来预期的好处，要看面向对象分析的

“正确性”。

让我们跳出面向对象设计的圈子，上升一层，考察包括 OOA 和 OOD 的一个更全局的视图，如图 22-8 所示。

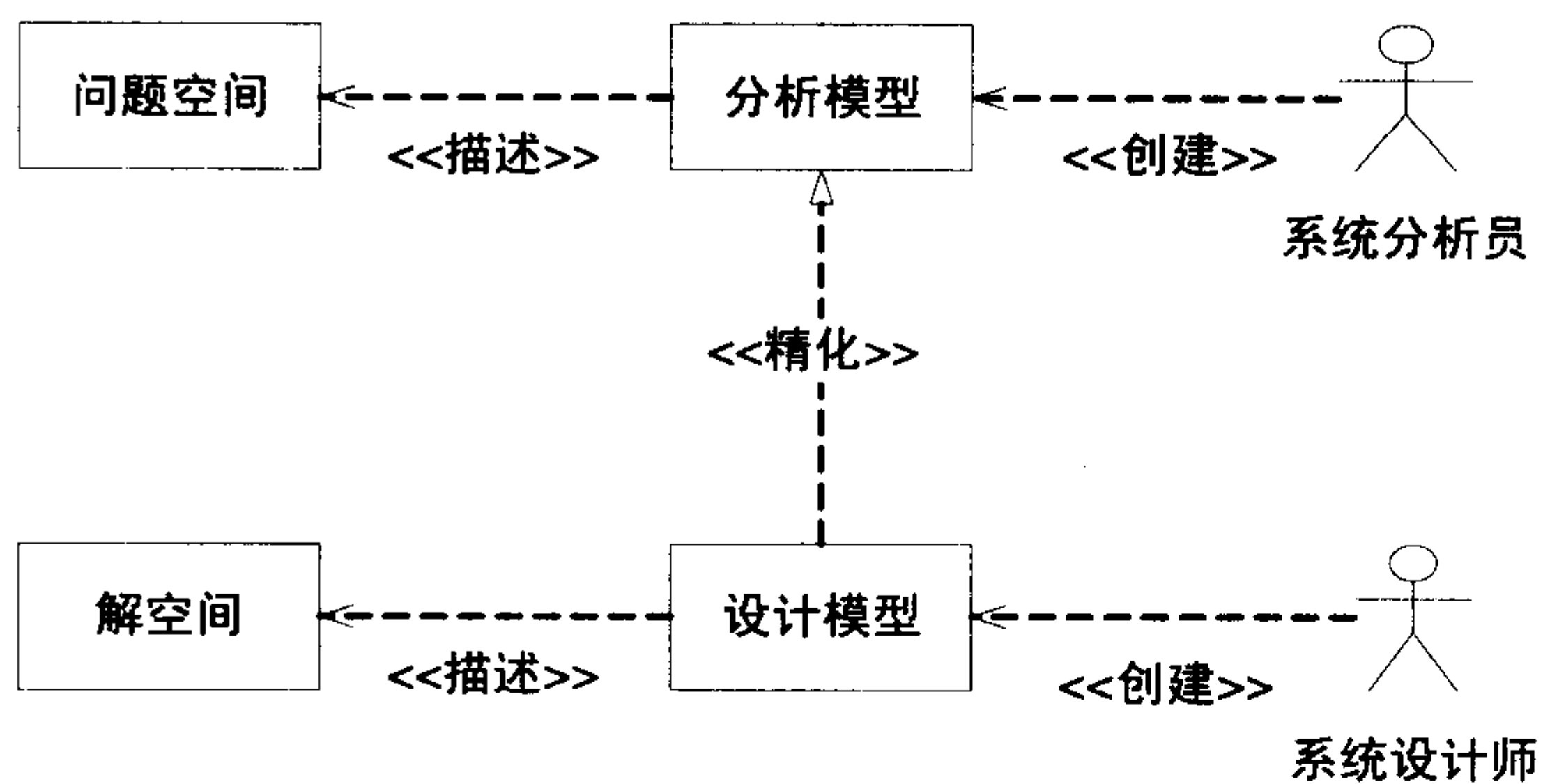


图 22-8 分析是设计的基础

分析模型是设计模型的基础，设计模型是分析模型的精化；我们对问题空间的认识的变化，首先影响分析模型，并间接对设计模型造成冲击；想要得到好的设计，首先要有好的分析。

套用一句常给编程人员说的那句“设计先行”，我想对设计人员说：“分析先行！”

## 第 23 章 超越设计模式：理解和运用更多模式

---

内行的设计者知道：不是解决任何问题都要从头做起。

——GOF,《设计模式》

当你看到 IBM 的广告“中间件就是 IBM 软件”时，你会产生片刻的困惑吗？如果答案是肯定的，那么你可以想想类似的两句话，或者说是两个思维定式。

- “可乐就是可口可乐”
- “模式就是 GOF 设计模式”

它们的共同特点在于，把一般的、涵盖范围更大的、往往也是更短的一个词，和一个特殊的、涵盖范围更小的、往往也是更长的词“等同”起来。从营销学的角度来讲，这叫“心理占位”。对此，广义营销领域的圣经《定位》里讲得明白：

进入人们大脑的捷径是，争当第一。

....

广告业正在进入一个战略至上的时代。在定位时代里，光靠发明或发现新东西是不够的，甚至可能没它也行；但你必须第一个打入预期客户的大脑才行。

GOF 所著的《设计模式》一书已成为一种“现象”。作为结果，越来越多的人开始学习和运用 GOF 设计模式。但同时，笔者注意到了另一个意想不到的结果，那就是有相当多的人下意识里认为“模式就是 GOF 设计模式”——很遗憾，这不是 GOF 打出的广告语，我相信他们也绝不希望结果会是这样。

正是这个“意想不到”的结果使本章的内容显得重要。毕竟，作为软件架构师，知识必须广博，所有软件领域的最佳经验都应成为他研究和借鉴的对象。

### 23.1 关于模式的两个问题

---

笔者曾遇到过两个问题，一个是关于模式的，还有一个也是关于模式的。

一个朋友说：“模式会影响性能。”这句话当然有问题。

还有一个问题，来自朋友的提问：“Remoting 模式和 GOF 设计模式有何关系？”

这两个问题的背后，隐藏着关于模式的一些比较普遍的“思维定式”：

- 一提起“模式”，就认为是“设计模式”
- 一提起“设计模式”，就认为是“GOF 设计模式”

这两种思维定式，像绳索一样将我们的思维捆得死死的。于是，对于分析模式、架构模式、测试模式、过程模式、配置管理模式、再工程模式、性能模式、Remoting 模式、反模式，等等，往往不能理解其“位置”；这样一来，模式本应发挥的作用就大打折扣了。

看来，我们必须超越设计模式，从更广泛意义上全面地认识模式，才能充分利用丰富的模式资源。

## 23.2 模式的正交分类法

### 23.2.1 正交思维

打破思维定式，还要靠正确的思维方法。对此，正交思维很有用。

简言之，如果两个或更多事物中的一个发生变化，不会影响其他事物，这些事物就是正交的。“正交性”是从几何学中借鉴而来的术语。如果两条直线相交成直角，它们就是正交的，比如图 23-1 所示的坐标轴。用向量术语说，这两条直线互不依赖——沿着某一条直线移动，投影到另一条直线上的位置不变。

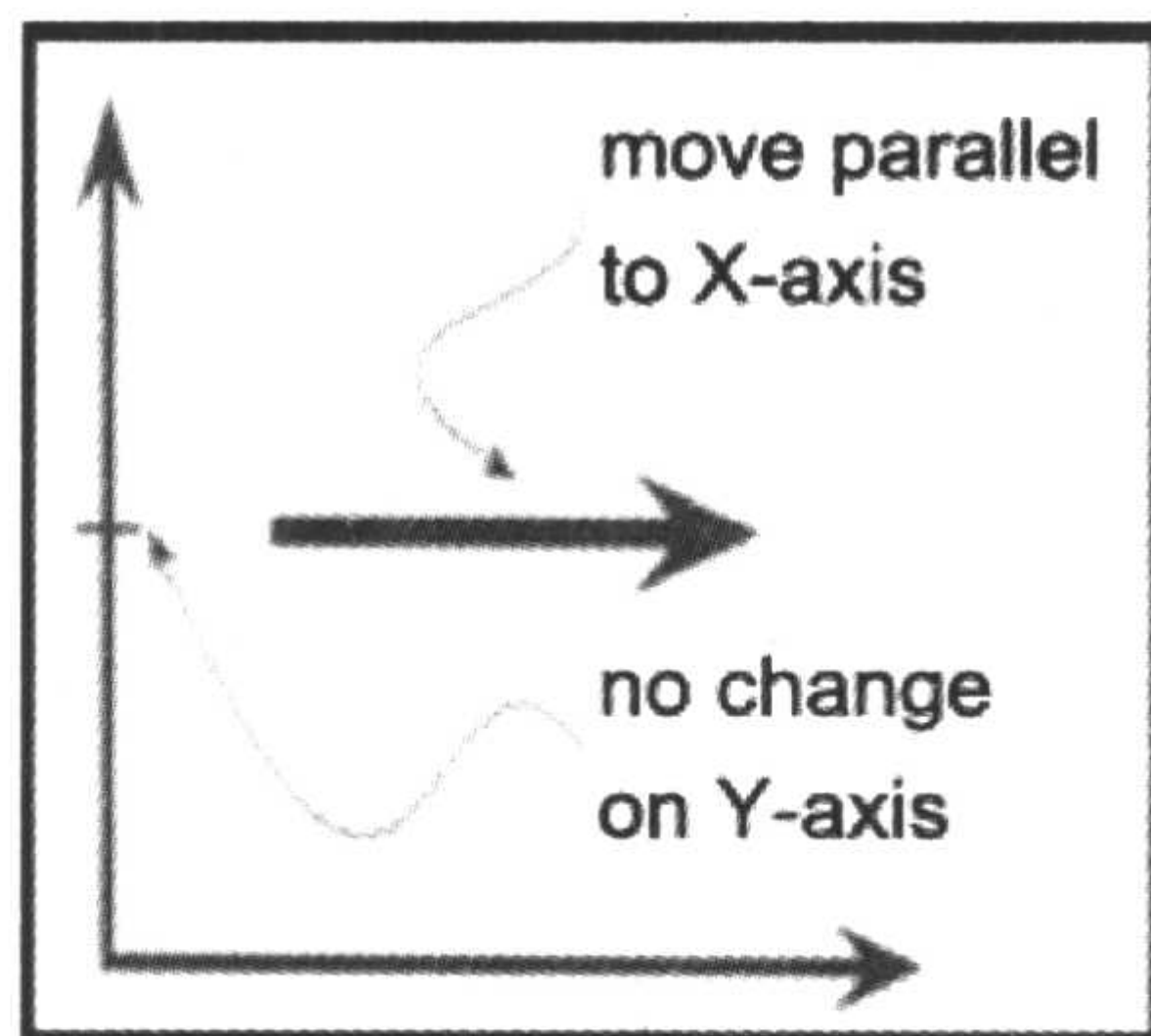


图 23-1 二维坐标系（图片来源：《程序员修炼之道》）

正交思维最有益的用处之一，是用于分类。图 23-2 中，显示了大小不一、形状不同、颜色各异的一些物体。如何分类呢？我们采用正交思维来分类！



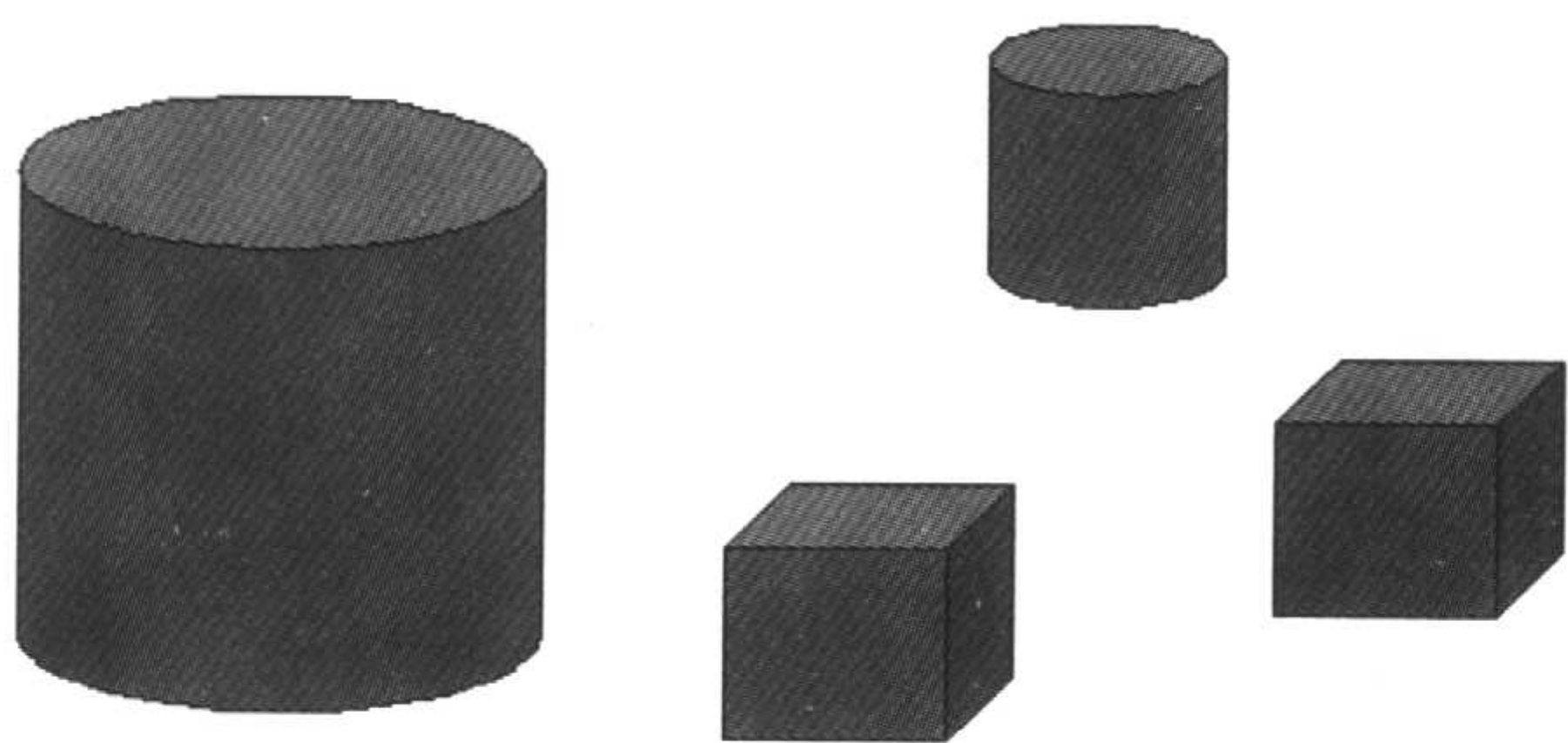


图 23-2 不胜枚举的物体

如图 23-3 所示，分别从形状、颜色、大小三个方面对物体进行分类。于是，每个物体在一种分类方法中，属于且仅属于一种类别。例如，红色的小正方体，其颜色为红、体积为小、形状为正方体。

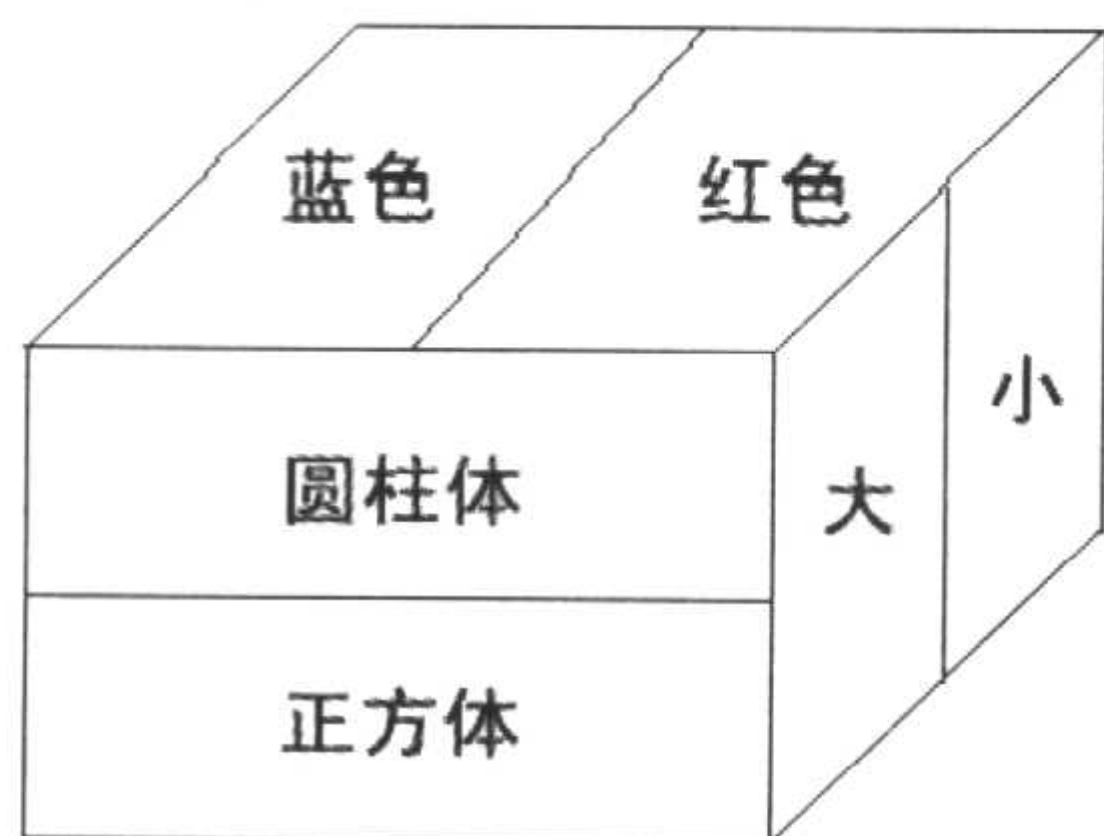


图 23-3 物体的三维正交分类法

其实，正交思维是“分而治之”思想的一种具体手段。要考虑的问题越复杂，正交思维的作用就越大。想象一下，图 23-3 的分类法中，包含更多的形状、更多的颜色、更细致的体积描述；而且，也可以不只三维，比如可以引入重量维、材质维等。

正交思维的作用，可以从“分析”阶段和“综合”阶段来刻画：思考的分析阶段，我们一次仅需专注于一个方面，使把握复杂的事物成为可能；思考的综合阶段，正交思维使你可以“优雅”地、多角度地、全面把握概念，充分收获“分而治之”的思维成果。

### 23.2.2 正交思维用于模式分类

回到我们的问题——模式怎样分类？图 23-4 表达了本章的观点，分别从三个独立的方面考察模式，最终得到模式的三维正交分类法：

- 这个模式为何项工作服务
- 这个模式是通用的，还是针对具体领域的
- 这个模式是应该推崇的，还是应该避免的





图 23-4 模式的三维正交分类法

软件开发会涉及工程、管理、支持等多方面的工作，而这些方方面面的工作都有各自的“经验总结”，这就是为什么模式能够（也应该）按“为哪项工作服务”来分类的原因。本章借鉴 RUP 九大工作流的概念，并对“分析与设计工作流”进行细化，将“工作”分为十一类；于是，相应的模式有：业务建模模式、需求模式、分析模式、架构模式、设计模式、代码模式、测试模式、部署模式、配置管理模式、项目管理模式、环境支持模式。值得说明的是，按“为何项工作服务”分类的思想是重要的，并不见得拘泥于本章的“十一种模式”的具体分类方案。读者根据自己的实际工作需要“修正”模式的三维正交分类法，是值得推崇的，例如关注 CMM 的读者可能需要“过程改进模式”等。

换一个角度。单从模式所针对的“领域”来看，模式的分类就不下几十种。因此，本章采用“笼而统之”的办法，将模式分为通用的、和领域相关的两种。例如，ERP 模式、供应链模式就是典型的“领域相关模式”；而 GOF 设计模式就属于“通用模式”，无论哪个领域的应用都可借鉴。

根据这个模式是应该推崇的还是应该避免的，又可将它分为“模式”与“反模式”两类。反模式与模式的概念类似，都是常见问题的可复用的经验总结，也都有名称、问题、解决方案、效果四个要素。但是，经验不一定正确，反模式就是对“不要那么做”的归档。为什么要总结反模式呢？原因之一是问题一出来，想当然地采取某种错误办法的情况屡见不鲜；而有了反模式的经验之后，则能够自觉避免。

### 23.3 专攻性能：性能模式简介

现在说说前面提到的“设计模式影响性能”的说法为什么有问题。不可否认，用设计模式“堆”出来的应用，照样可能是性能不佳的；但是，这并不能说明设计模式是性能不佳的根源。

进攻是最好的防守——下面介绍性能模式。

《软件性能工程》一书介绍了 7 个性能模式，它们分别为：

- 替代路由。对高使用率的对象，分散对其的请求到不同的对象或位置；
- 弹性时间。分散对高使用率的请求到不同时间；
- 批处理。把请求组合成批，从而使开销的处理只需执行一次而不是每个请求都执行一遍；
- 耦合。让接口与最频繁使用的对象匹配；
- 快速通道。确定支配性工作负载功能，并简化其处理过程，只剩下必要的部分；
- 弱化周期性功能。最小化必须按固定间隔执行的工作量；
- 重要事情优先。专注于相对重要的处理任务；如果每个任务都不能在可能的时间内完成，则保证被忽略的是最不重要的任务。

这些性能模式是专门解决性能问题的架构级和设计级的模式，它在模式分类“坐标”中的位置如图 23-5 所示。



图 23-5 性能模式的“位置”

可贵的是，书中并不是孤立地谈性能模式，而是基于性能问题的全面分析，高屋建瓴地归纳了 9 条性能原则。例如：

- 处理与频率。使处理与频率的乘积最小；
- 分散负载。有可能在不同时间或不同位置处理冲突的负载时，分散这些负载；
- 本地化。创建与物理计算机资源接近的活动、功能和结果；
- 中心化。确定支配性工作负载功能，并使之处理过程最小化。

每个性能模式，都遵循一条或多条性能原则的指导思想。图 23-6 总结了性能模式与性能原则的关系。该图采用 UML 语法，性能模式遵循性能原则用 UML 的“实现”关系表示。

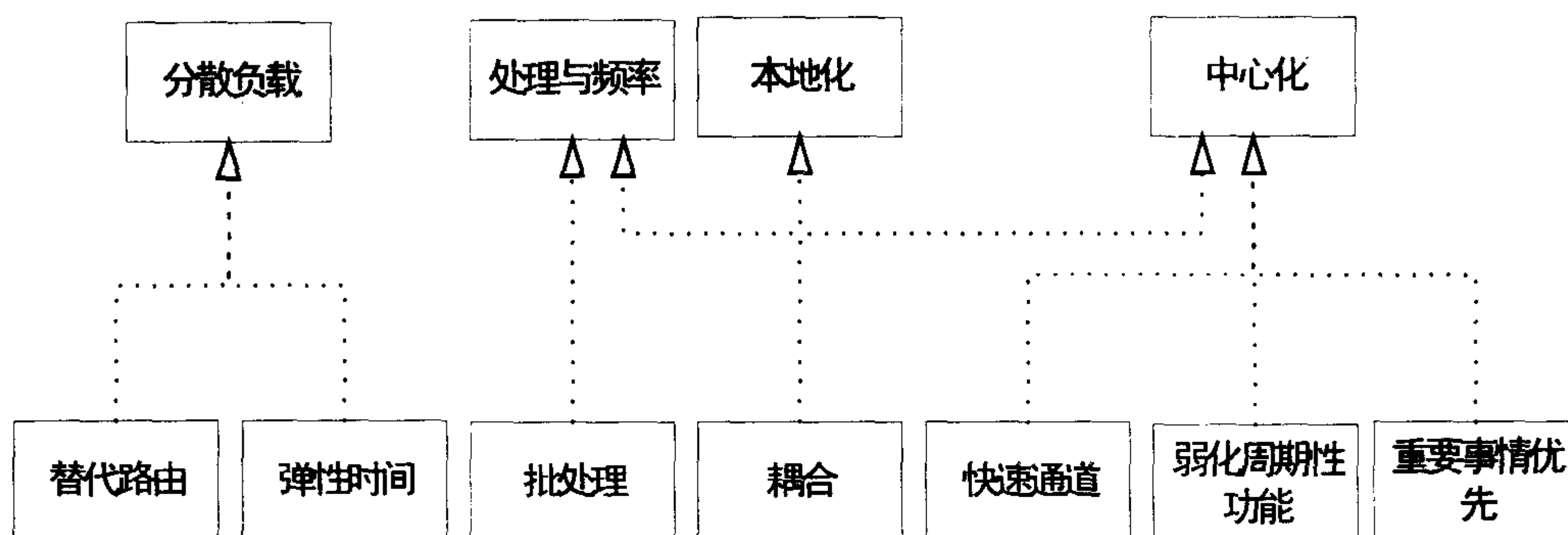


图 23-6 性能原则与性能模式

最后，我们以批处理模式为例，说明性能模式是如何提高性能的。批处理模式遵循“处理与频率”性能模式的思想，追求“处理与频率的乘积最小”。批处理模式的要点如下：

- 问题。对于频繁执行的任务，系统额外的处理时间可能比真正的处理时间还多；
- 解决方案。将请求合并成批处理，如图 23-7 所示；
- 效果。批处理模式减少了全部任务所需的总处理量，对开销的处理量和请求频率都很高时非常奏效，反之则应慎重应用。

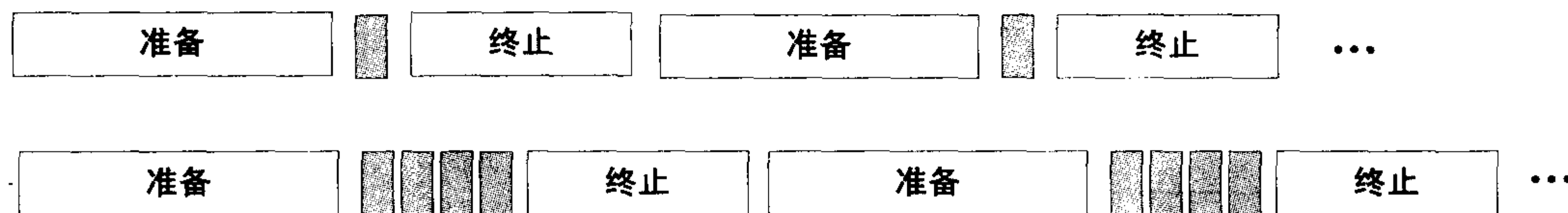


图 23-7 批处理与单独处理的比较（图片参考：《软件性能工程》）

至此，我们已经了解了性能模式。“设计模式影响性能”的说法有两点欠妥：第一，模式之于性能，应当用两分法来看；第二，对专门解决性能问题的模式缺乏了解。



## 23.4 模型驱动开发的方方面面：MDD 模式简介

下面，简介 MDD 模式——它是一套涉及面更广的综合模式——从而加深对模式的三维正交分类法的认识。

模型驱动开发（Model-Driven Development, MDD）是业界的大趋势。在经历了代码可视化、双向工程之后，我们有理由相信纯 MDD 终有一天会成为主流。对此，Markus Voelter 等人的论文《Patterns for Model-Driven Development》做了前瞻性的对 MDD 模式归纳工作，对平台级软件的开发商很有指导意义。

比如，正如 GOF 设计模式分为创建型模式、结构型模式、行为型模式，MDD 模式被分为过程和组织模式、领域建模模式、应用平台开发模式，如图 23-8 所示。

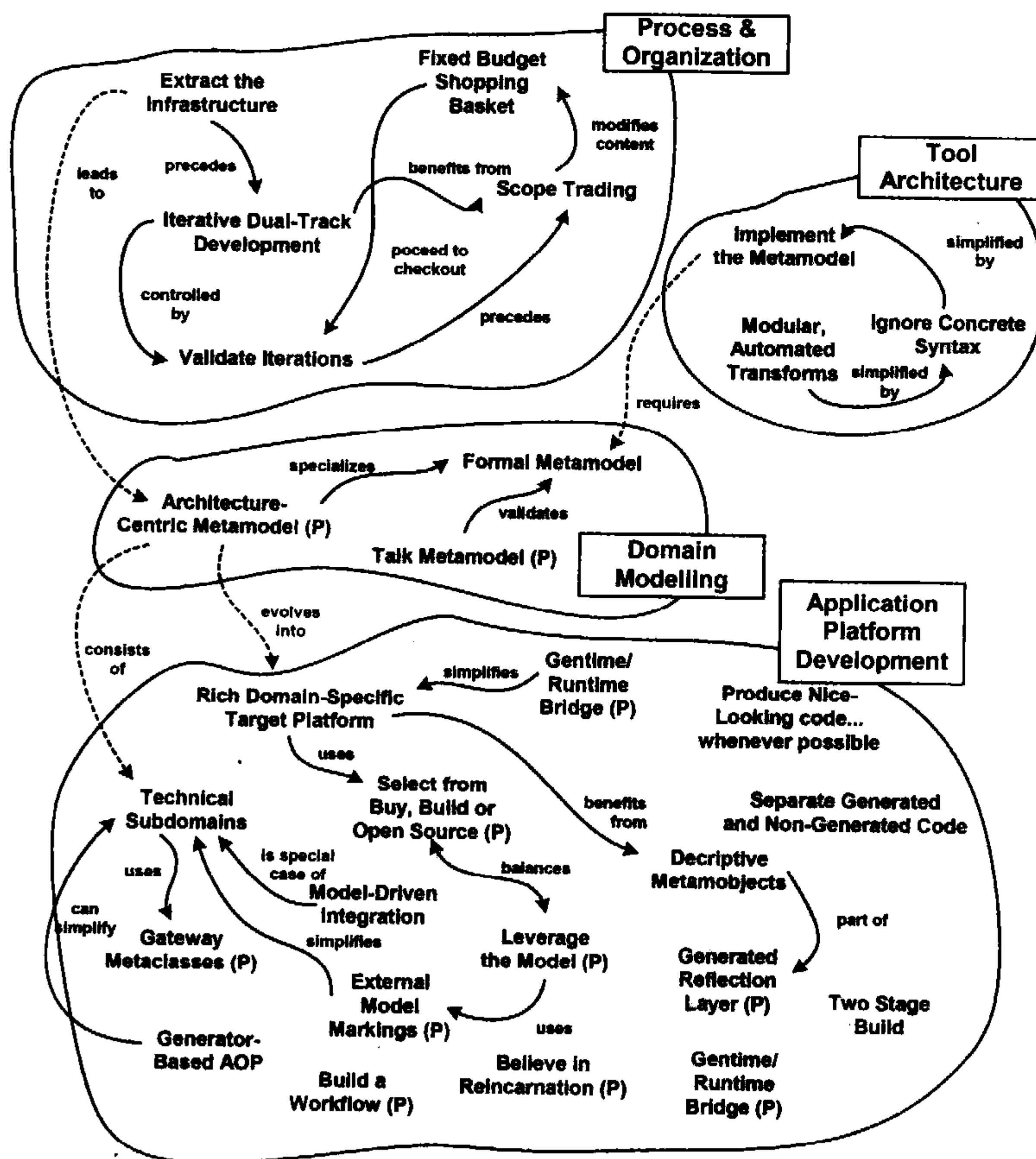


图 23-8 MDD 模式（图片来源：《Patterns for Model-Driven Development》）

MDD 模式的归纳工作还在进行过程之中,《Patterns for Model-Driven Development》公布的 MDD 模式也只是阶段性成果,但其覆盖面也已经比较广了。这些 MDD 模式在模式分类“坐标”中的位置如图 23-9 所示。

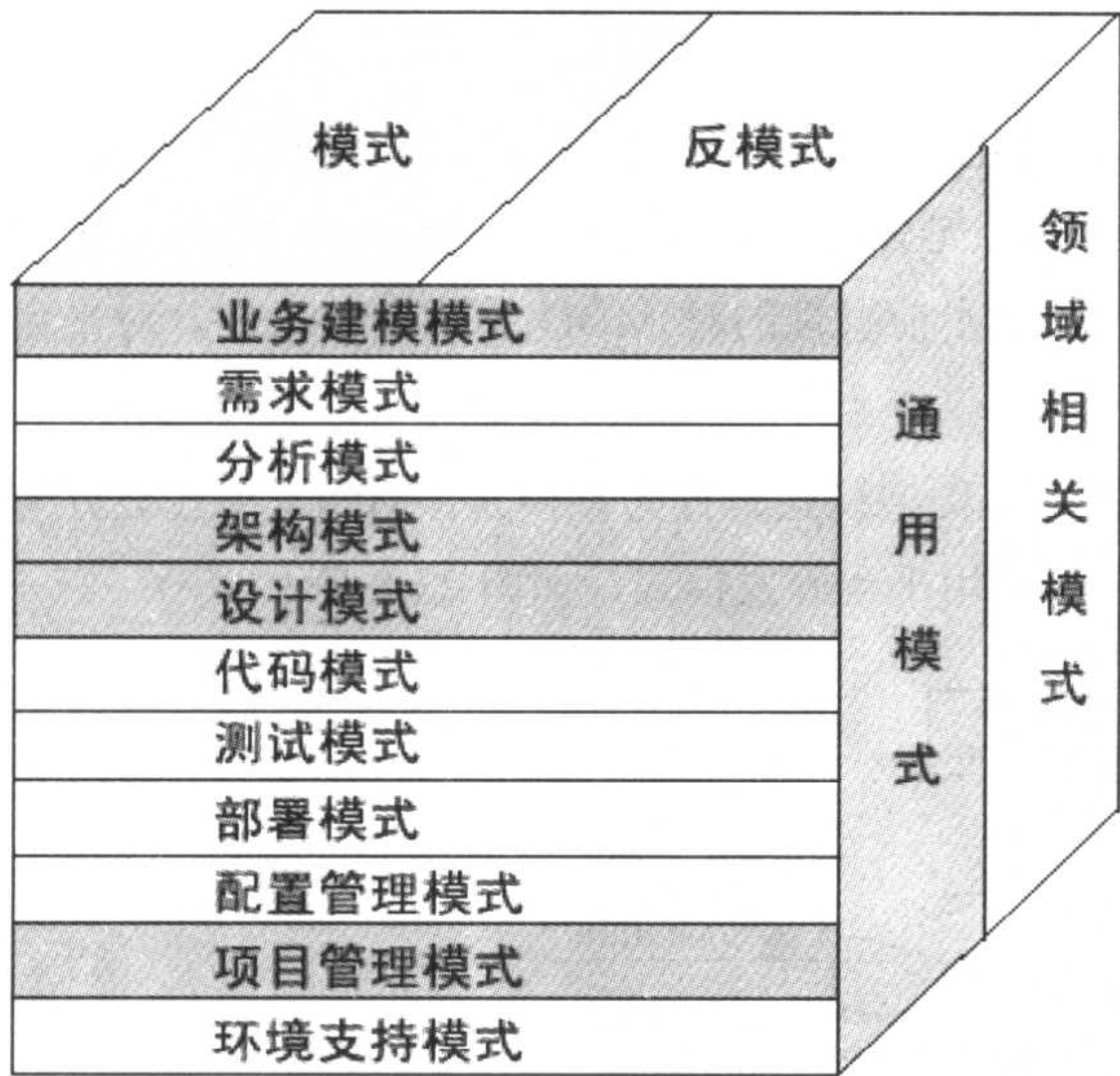


图 23-9 MDD 模式的“位置”

### 23.5 总结：拥抱模式

英国数学家怀特海说过,“模式具有重要性的看法和文明一样古老。每一种艺术都奠基于模式的研究。”软件学科虽然还很年轻,但已经积累了大量宝贵的模式,并且,这种趋势还在加强!

希望每个人对模式这种“经验的结晶”都给予应有的重视,也希望本章介绍的“模式的三维正交分类法”对读者“接纳”更多模式有所帮助。

## 第 24 章 如此轻松：立足图论学 UML

---

UML 的最终目标是在尽可能简单地同时能够对实际需要建立的系统的各个方面建模。

——James Rumbaugh, 《UML 参考手册》

对很多事情的处理上，东西方都大相径庭。究其根底，往往是东西方文化的差异使然。“有工具的傻子还是傻子！（A Fool with a Tool is Still a Fool!）”这句在软件工程界颇为有名的话，就体现了西方人说话不大客气的特点。作为中国人，你可能不大喜欢这句话的表达方式，但其内容的正确性是不容置疑的——它强调了工具背后的思想才是最重要的。

学习和应用 UML，把握 UML 背后的思想才是最重要的，因为 UML 本质上也是一种辅助思考的工具。UML 作为 OMG 组织认可的一种标准化的可视化建模语言，近年来在国内外软件界非常盛行——相关的书籍不可谓不多，学习和使用 UML 的人不可谓不众。但是，笔者发现不少人犯了买椟还珠的毛病——他们忘记了 UML 只是我们表达建模思想的工具，其背后的思想才是最重要的。

心理学的研究也表明了这一点。在心理学中，“语言”和“言语”关系紧密：

- “语言”是一种由词汇和语法构成的符号系统，人们使用语言进行思想交流则称为“言语”
- 言语可以分为三种形式：口头、书面，以及内部言语
- 语言是思维的基础，并对思维具有反作用
- 思维对事物的反映总是借助语言进行的，思维过程通过内部语言进行，思维结果通过口头或书面语言表现

作为软件架构师，不仅应该掌握 UML 建模技术，还应该努力达到“通过建模促进思维”的境界——这在软件规模越来越大的今天是很有现实意义的。为此，本章将结合实例，说明图论思想在 UML 应用中的意义。

## 24.1 管窥 UML 中的 OO 思想

这是个真实的故事。通过它，希望更多的人了解 UML 背后的思想比它的语法更重要。

### 24.1.1 一道笔试题的故事

这道笔试题是这样的。

写出下列程序的运行结果：

```
public class Test {
    public static void main(String[] args) {
        Child child = new Child();
    }
}
class Parent {
    Parent() {
        System.out.println("to construct Parent.");
    }
}
class Child extends Parent {
    Child() {
        System.out.println("to construct Child.");
    }
    Delegatee delegatee = new Delegatee();
}
class Delegatee {
    Delegatee() {
        System.out.println("to construct Delegatee.");
    }
}
```

这道题很简单，就是考构造函数的执行顺序，输出结果如下：

```
to construct Parent.
to construct Delegatee.
to construct Child.
```

然而，应试者没有写出程序运行结果，倒是画了一张类图出来（如图 24-1）：

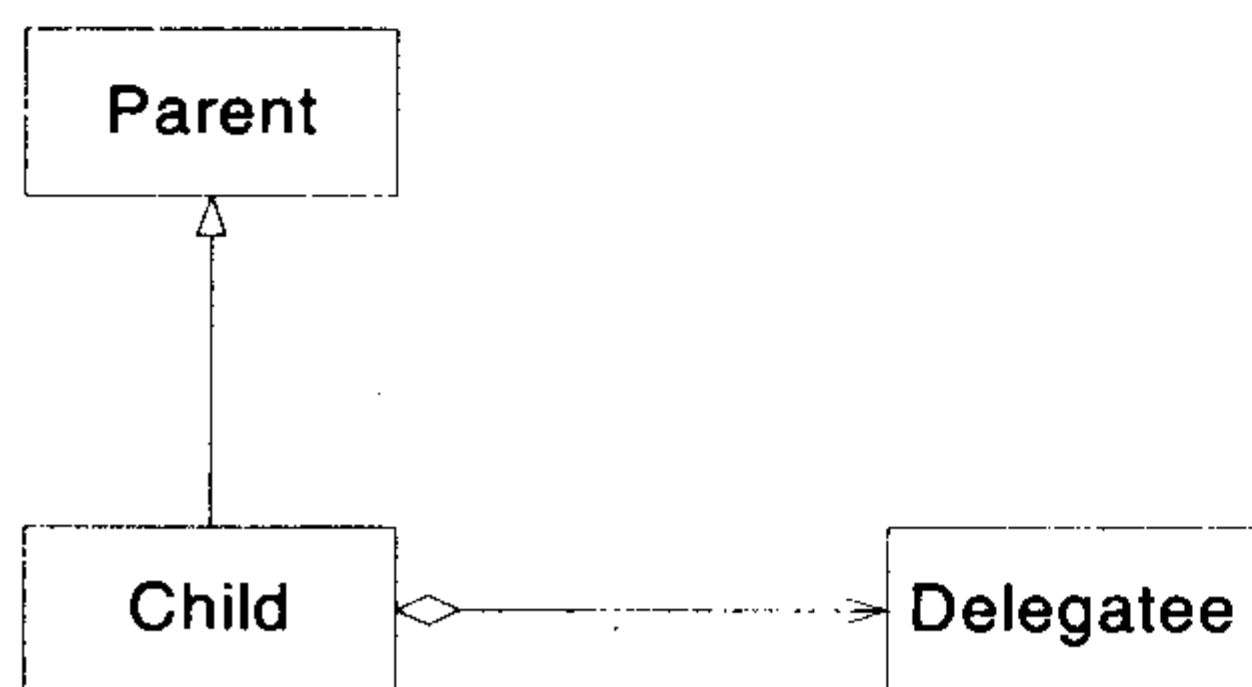


图 24-1 程序对应的类图



我问他：“你画的类图很好，用 UML 几年了？”

他说：“2 年多了。”

我说：“为什么不写程序运行结果呢？太简单了吗？”

他嘿嘿一笑，说：“这道题考的是构造函数的执行，我记不大清了。”

我说：“构造一个类时，它的父类和成员变量所属类的构造函数都会被自动执行，具体顺序是先 Parent 后 Delegetee 最后 Child，是吧？”

他说：“对对对。不过，我不喜欢这些细节性的东西，还是图形化的类图更能表达程序的思想。”

我说：“是呀，那么请你谈谈这个简单的类图中的思想好吗？”

他无言。

### 24.1.2 UML 背后的思想

真是可惜，我本以为他会说出上面类图中的依赖思想呢！UML 图里真的充满了思想，哪怕是上面那个如此简单的类图！

继承的箭头为何指向父类？

相信至今还有人会画出类似图 24-2 的类图，小小的一个错误却叫人不禁扼腕！其实，大师们把继承的箭头方向规定为指向父类，是有深刻的设计思想的——它代表了依赖的方向！

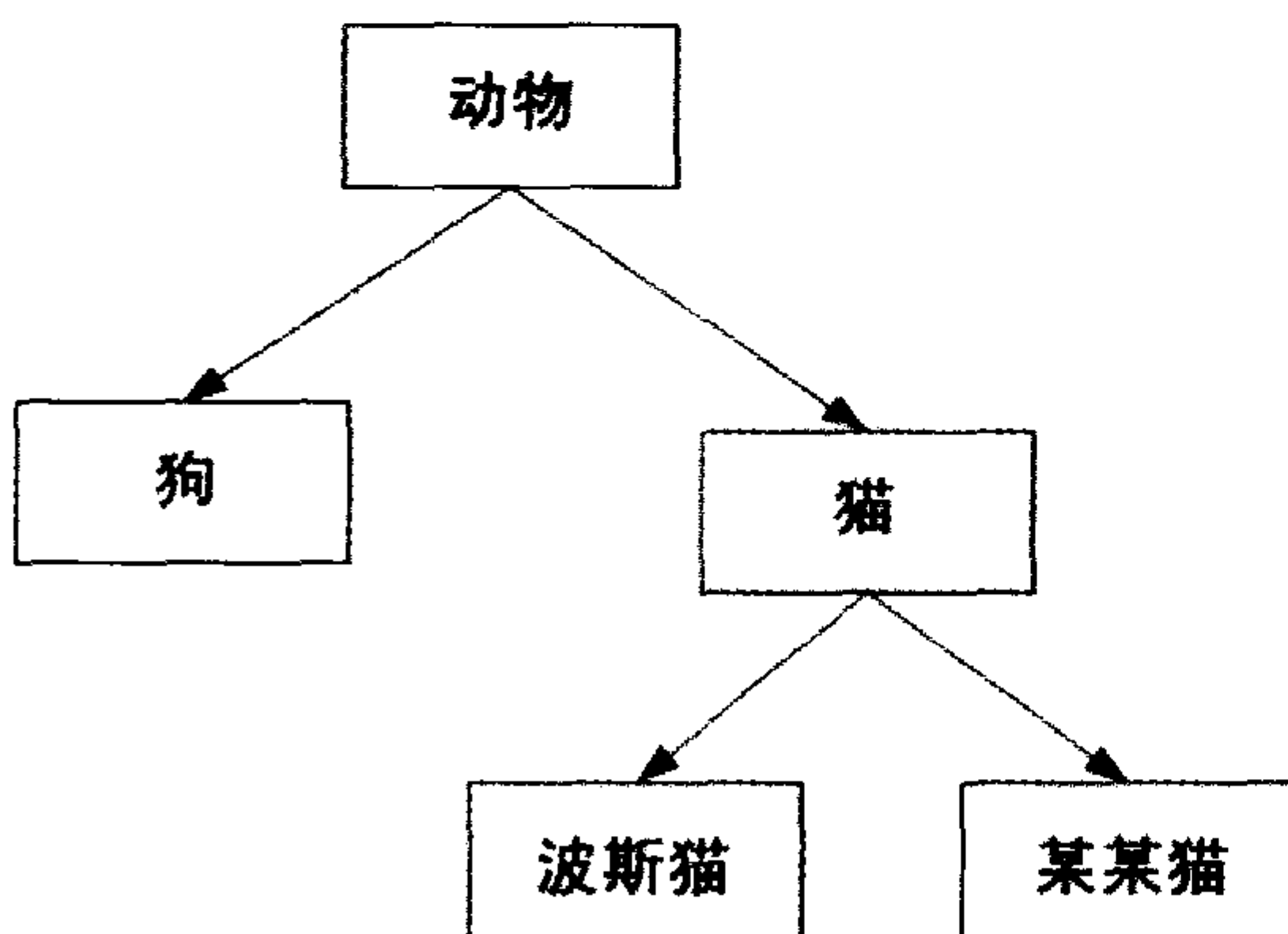


图 24-2 错误的类图

所谓依赖（Dependency），是指两个元素之间的一种关系，其中一个元素变化，导致另一个元素变化。UML 中采用从子类指向父类的空心箭头表示继承，暗示父类的变化可能导致子类的变化。

再进一步，回到笔试题。其实，构造函数的执行顺序，何尝不是唯“依赖关系”马首是瞻呢？具体而言，就是“被依赖的先构造，依赖于人的后构造”。如图 24-3 所示，类 Child 继承自类 Parent 意味着前者依赖后者，而 Child 对 Delegatee 的聚集关系也意味着类似的依赖关系。

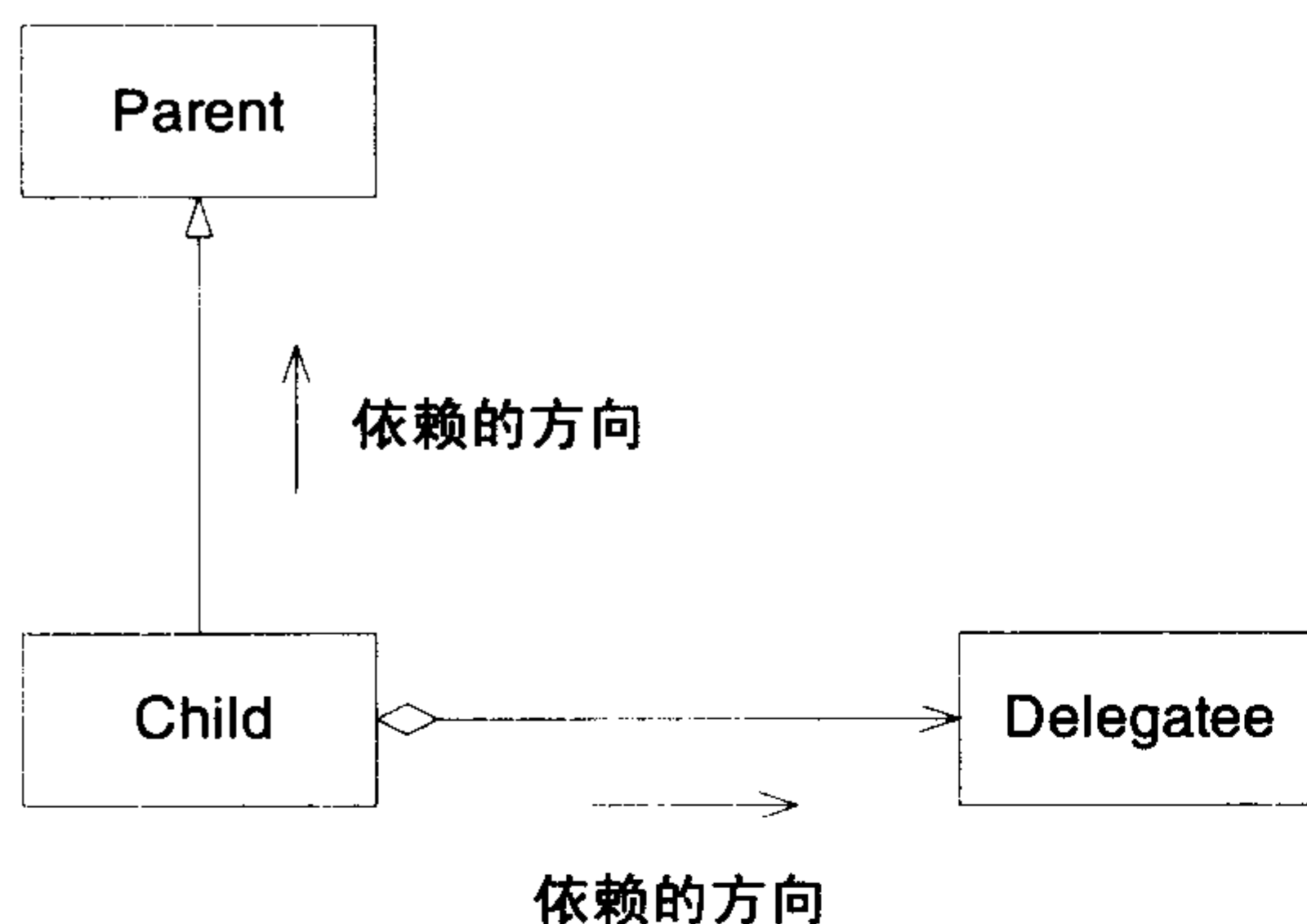


图 24-3 被依赖的先构造，依赖于“人”的后构造

通过上面这个简单的例子，对“UML 背后藏着许多 OO 思想”这一点我们算是管窥之一斑了。本章的以下部分，将从图论的角度讨论 UML，并会论及更多 UML 背后的 OO 思想。

## 24.2 图的定义与 UML 应用

### 24.2.1 图的定义

顾名思义，图论就是研究图的理论。图是一种由两个集合——即一个顶点集合和一个边集合——定义的抽象数据结构。图的更形式化的定义如下：

称  $G = (V, E)$  是一个图，如果

- (1)  $V$  是一个非空有限集合，
- (2)  $E$  是  $V$  中元素的无序对所组成的有限集合，

并把  $V$  的元素叫做图的顶点， $E$  的元素叫做图的边。

举个例子，图 24-4 展示了一个有 7 个顶点和 5 条边的图， $v_i$  标出了顶点， $e_i$  标出了边。

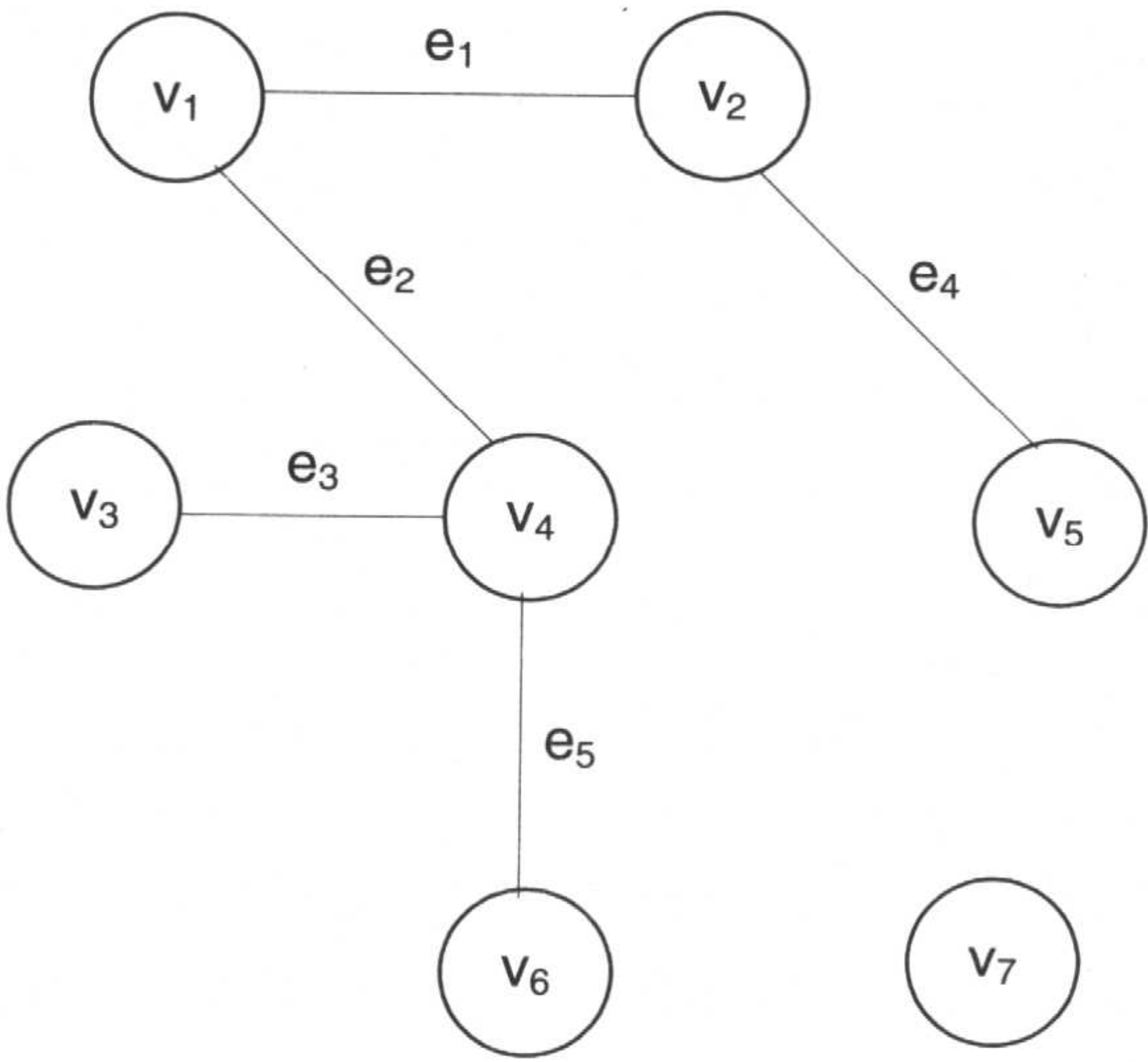


图 24-4 一个有 7 个顶点和 5 条边的图

24.2.2 图的定义的 UML 应用——UML 的图论观点

UML 作为可视化建模语言，包括语法和语义两个方面。单从语法方面，用图论的眼光——把 UML 看作顶点和边——来学习 UML，应当说是正本清源之道。表 24-1 以图论观点对 UML 语法进行了总结。

表 24-1 立足图论，总结 UML 语法

	顶点	边	边属性	其他
用例图	参与者， 用例	关联，泛化， 包含，扩展		接口
包图	包， 接口	依赖， 实现		可嵌入类图
类图	类	关联， 泛化， 依赖	角色名，多重性，导航， 组成符，聚集符，关联 名，关联名方向	限定符， 参数化类， 关联类
对象图	对象	链	角色名，多重性，导航， 组成符，聚集符，链名， 链名方向	
顺序图	对象	消息	消息名，条件，重复	参与者实例，生命线， 激活

	顶点	边	边属性	其他
协作图	对象	链， 消息	消息号，所有顺序图的边 (消息)属性，所有对象图 的边(链)属性	参与者实例，位置，状 态，变成流，拷贝流
构件图	构件，接口	依赖		可嵌入对象图
部署图	节点	连接		可嵌入构件图
状态图	状态	转换	条件，动作	复合状态
活动图	活动状态	完成转换	条件，分支，分叉，结合	泳道，对象流

数学中，有关“数学抽象度”的研究表明：抽象层次越高，切近事物本质越深。UML 的图论观点，从更抽象的“图论”角度理解 UML 的语法，因此能够“切近事物本质更深”。UML 2.0 正被人们慢慢接受，改动虽大，但决不会跳出图论范畴；总之，理解了 UML 的图论观点，对快速掌握 UML 新规范大有裨益，笔者的实践也证明了这一点。

24.2.3 图的定义的 UML 应用——关联类语法的理解

除了上面的基本总结以外，笔者发现 UML 中的关联类常被“误用”或“该用不用”，所以有必要谈一下。

语法方面，从图论中对图的基本定义，可以找到对关联类的“犀利”的理解。  
首先，扩展一下图论的“经典”定义，如图 24-5 所示。

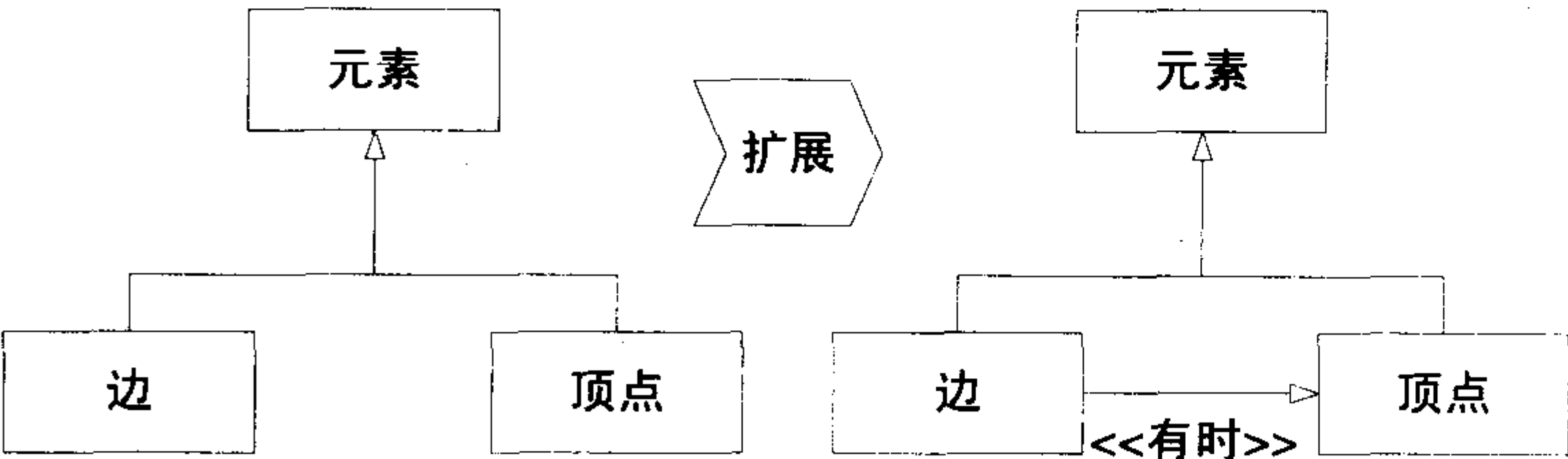


图 24-5 边也可以充当顶点

扩展之后，顶点可以由更多的“角色”来承担：除了通常的顶点外，边也可以充当顶点。这样以来，边就有如下三种情况：

- 连接顶点与顶点的边
- 连接顶点与边的边
- 连接边与边的边

然后，分析关联类本身的语法，它用到了上面扩展的第二种情况。如图 24-6 所示，关联类语法分为关联部分、类部分、关联部分和类部分之间的可视化连接部分，共三部分内容。



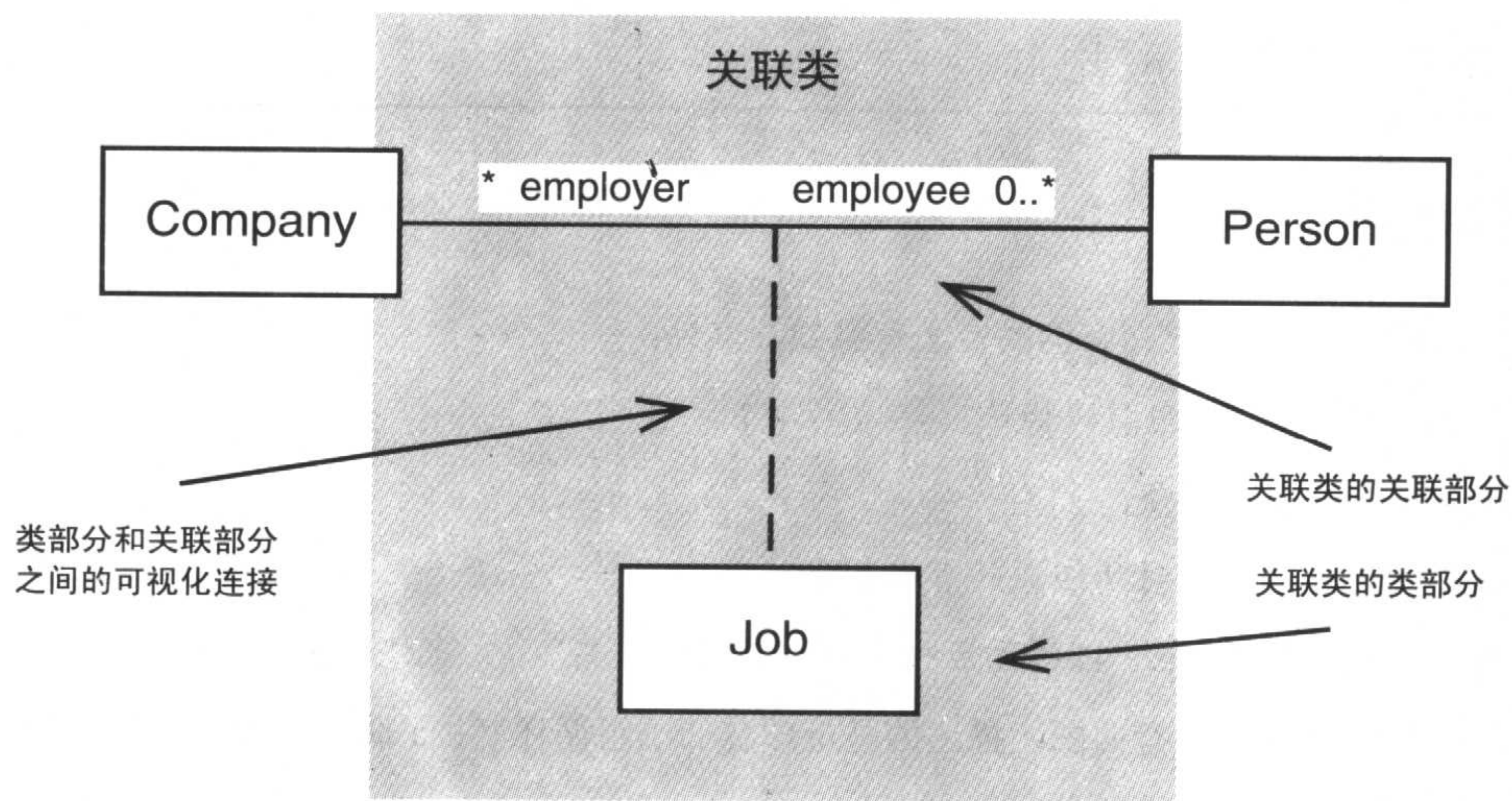


图 24-6 关联类语法解析

总之，虽然从语义来讲，关联类是一个独立的模型元素，但从语法角度，它既包含了关联的符号，又包含了类的符号。

#### 24.2.4 图的定义的 UML 应用——说说序列图

值得补充说明的是，序列图中“生命线”和“激活”也是可以充当顶点的边，如图 24-7 所示。

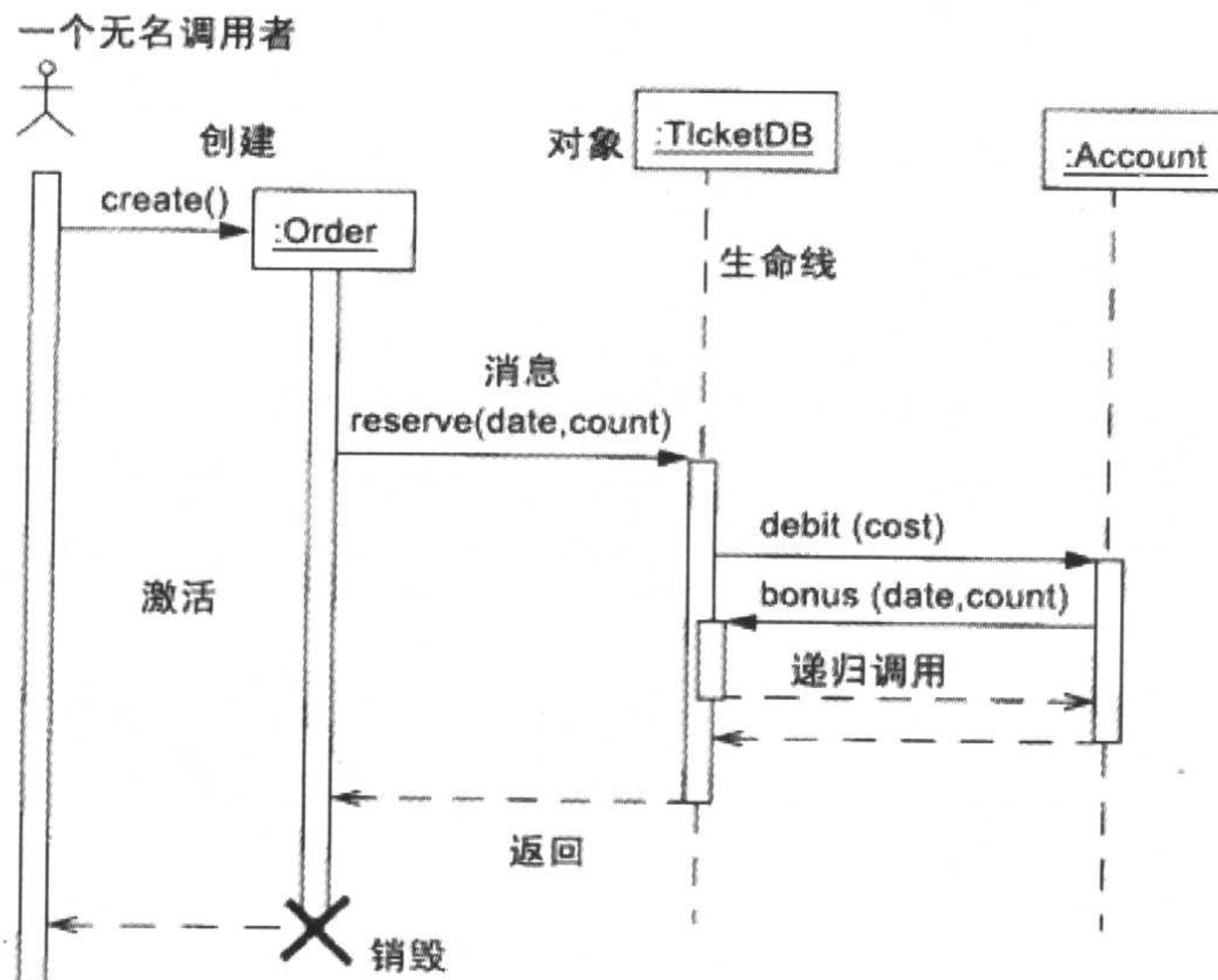


图 24-7 序列图一例（图片来源：《UML 参考手册》）

## 24.3 有向边与 UML 应用

### 24.3.1 有向边

有向图是无向图的特殊情况，它们的定义有微妙的差异。

称  $G = (V, E)$  是一个有向图，如果

(1)  $V$  是一个非空有限集合，

(2)  $E$  是  $V$  中元素的有序对所组成的有限集合，

并把  $V$  的元素叫做图的顶点， $E$  的元素叫做图的有向边。

举个例子，图 24-8 是一个有向图，描述的是足球赛的小组循环赛：a 队 3 场全胜，b、c、d 这 3 个队都是 1 胜 2 负。

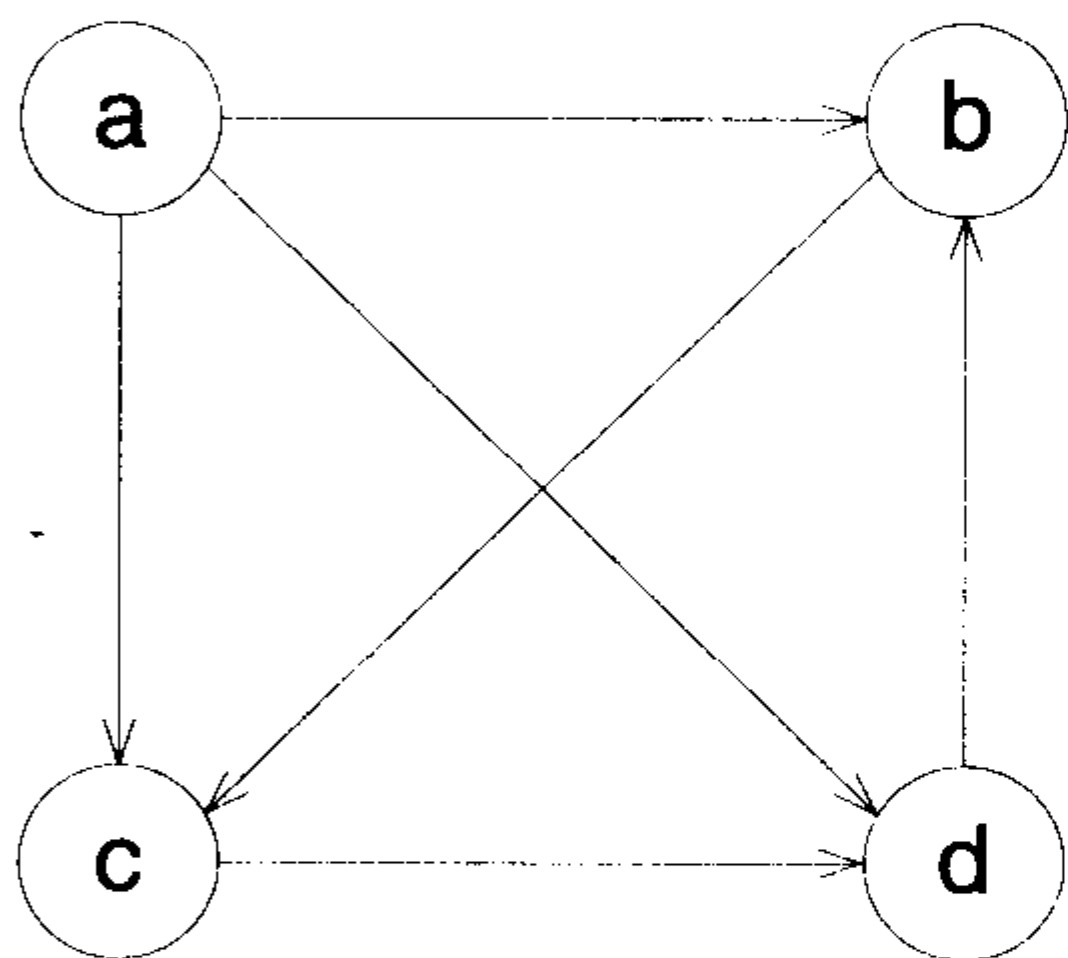


图 24-8 小组循环赛结果

### 24.3.2 有向边的 UML 应用——依赖关系

静态视图是 UML 的基础。模型中静态视图的元素是应用中有意义的概念，这些概念包括真实世界中的概念、抽象的概念、实现方面的概念和计算机领域的概念。静态视图中的关键元素是类元及它们之间的关系；类元是描述事物的建模元素，包括类、接口和数据类型等；类元之间的关系有依赖、泛化、实现和关联等。

依赖表示两个或多个模型元素之间语义上的关系，即提供者的某些变化会要求或指示依赖关系中客户的变化。例如，图 24-9 体现了一条架构设计的基本原则：问题领域层“不依赖于”其他任何层，而其他任何层“只依赖于”问题领域层。



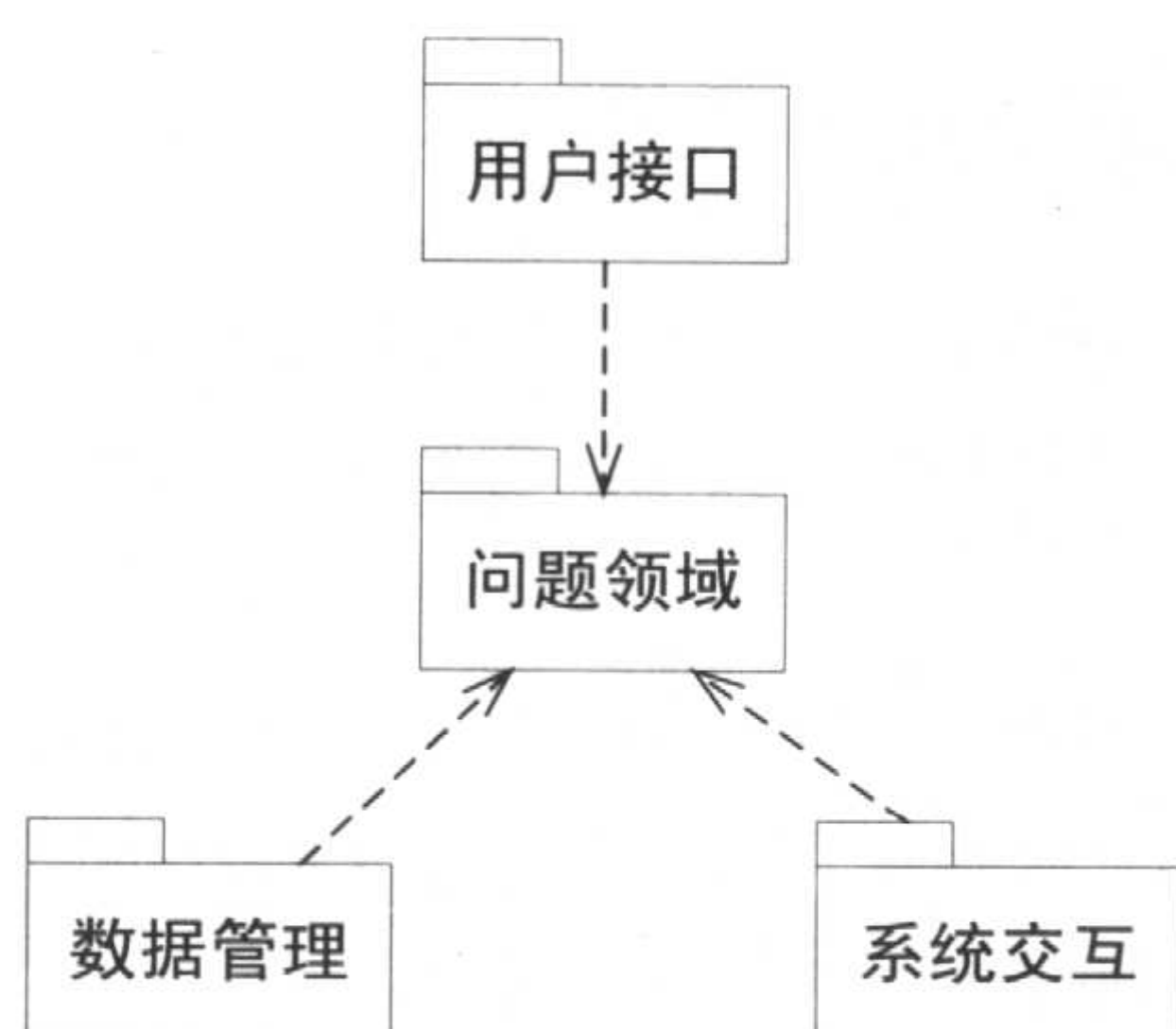


图 24-9 依赖的理想化处理原则

再举一例。图 24-10 是著名的观察者模式的一个变体，我们研究一下这个例子中的继承关系。Broadcaster 和 Listener 是 Framework 层的两个类，负责完成我们在开发的前期就抽象出来的“订阅—通知”机制。而 ConcreteBroadcaster 和 ConcreteListener 是使用“订阅—通知”机制的类，是通过继承实现的。显而易见，依赖倒置原则规定的“细节依赖于抽象”，在这里表现为“继承的箭头从子类指向父类”。

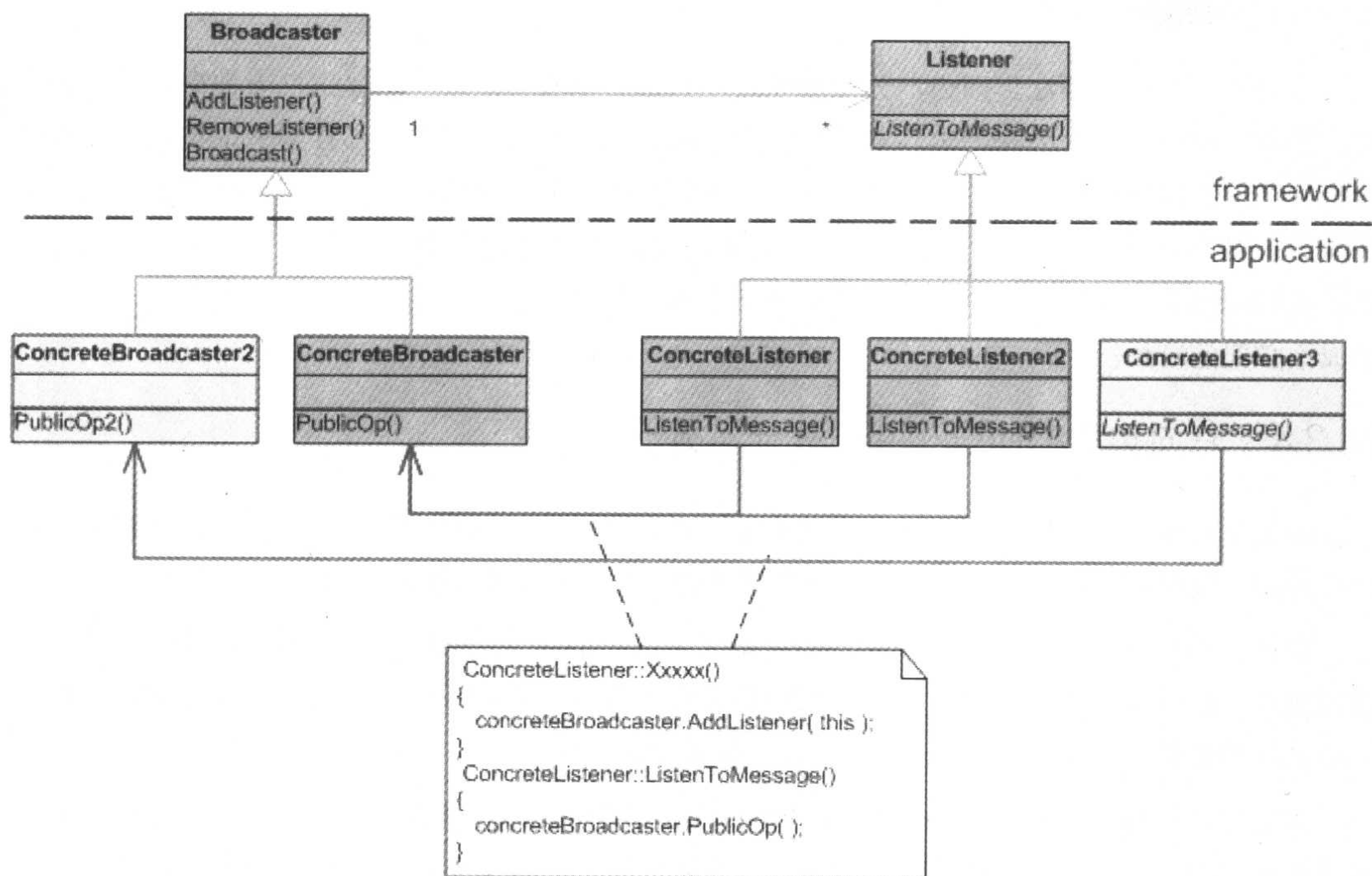


图 24-10 依赖方向与应用框架

### 24.3.3 有向边的 UML 应用——泛化、实现和关联的依赖思想

根据依赖关系的定义——依赖表示两个或多个模型元素之间语义上的关系，即提供者的某些变化会要求或指示依赖关系中客户的变化——泛化、实现和关联也是依赖关系，它们都包含了依赖的思想。

下面借助“关联关系中包含着依赖思想”这一认识，来说明领域建模中的一个不太规范的画法容易造成问题。在进行领域建模的时候，比如我们要表达“公司雇用人”这一语义时，不同的人可能画出的类图并不相同，如图 24-11 所示。那么，哪一种画法更值得推崇呢？

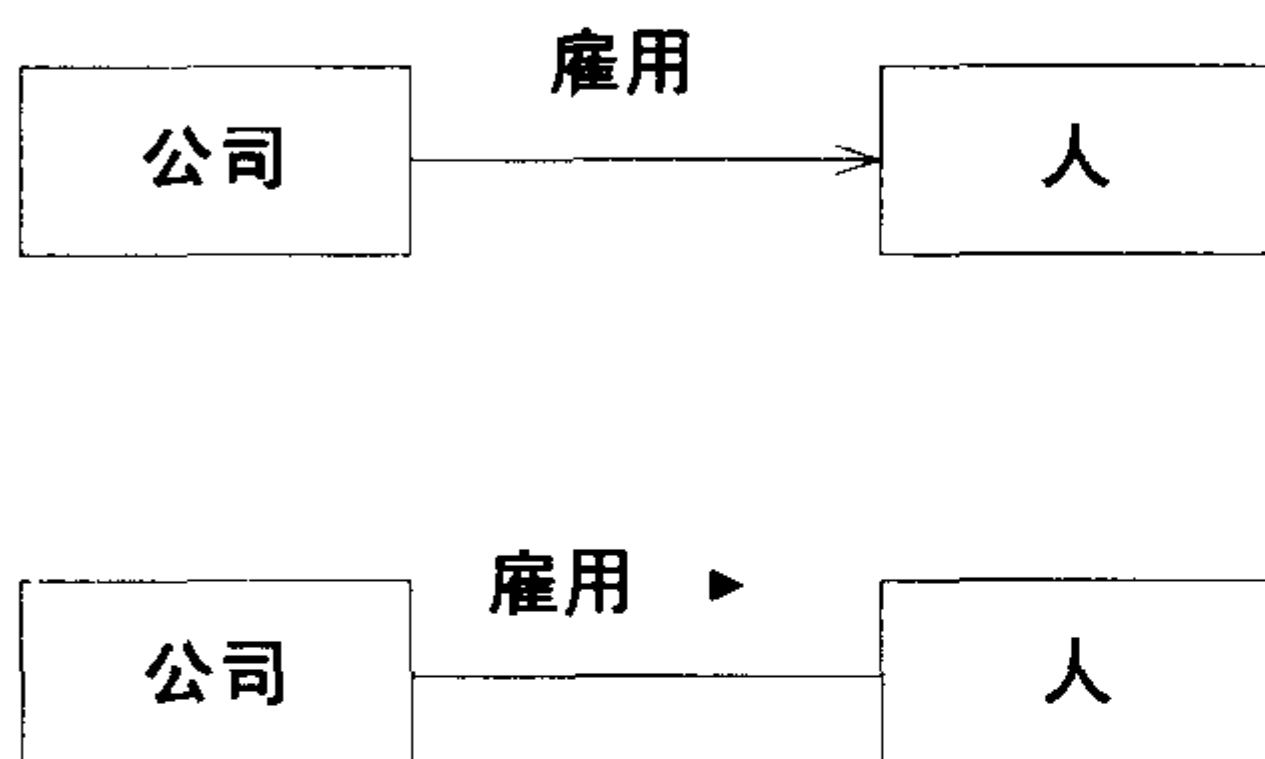


图 24-11 领域建模一例的两种画法

本书推荐第二种画法。因为，公司雇佣人，只是意味着“雇佣”的“语义方向”是从公司到人，并不一定意味着“公司”类到“人”类的单向关联。像第一种画法那样建模，在领域建模阶段自然是没有问题的；但到了后面设计之时，领域建模需要被精化成设计模型，此刻看到这幅类图的设计师会产生歧义：这个评审通过的类图是要说明“公司”类持有“人”类的引用吗？因此，我们应该从一开始就区分“关联名的语义方向”和“关联本身的方向”之间的不同，以避免造成不必要的麻烦。

### 24.3.4 有向边的 UML 应用——一个例子

图论是对现实世界中实际问题的高度抽象。有向图可以对具有特定依赖关系的实体群落，进行有效的抽象刻画，使人们能够忽略无关紧要的众多细节，而牢牢把握住本质性的依赖关系。

依赖关系在软件开发中的重要性，不管怎么强调都不过分。响应变化的能力往往决定一个项目的成败，而依赖关系的处理（其实不仅包括类与类等工件之间的依赖，还包括人之间的依赖和活动之间的依赖，详细讨论参见第 22 章）正是其中的关键。

著名的开放封闭原则（Open-Closed Principle, OCP）规定，“软件实体应该是可以扩展的，但是不可修改”。本原则紧紧围绕变化展开，变化来临时，如果不必改动软件实体的源代码，就能扩充它的行为，那么这个软件实体的设计就是满足开放封闭原则的。如果我们预测到某种变化，或者某种变化发生了，我们应当创建抽象来隔离以后发生的同类变化。在 Java 中，这种抽



象指抽象基类或接口；在 C++ 中，这种抽象是指抽象基类或纯抽象基类。

比如，在开发一个需求跟踪工具的时候，起初可能仅需要支持保存为专有格式的“项目”文件，但后来又需要支持导出为 HTML 格式的网页。最终的设计如图 24-12 所示。

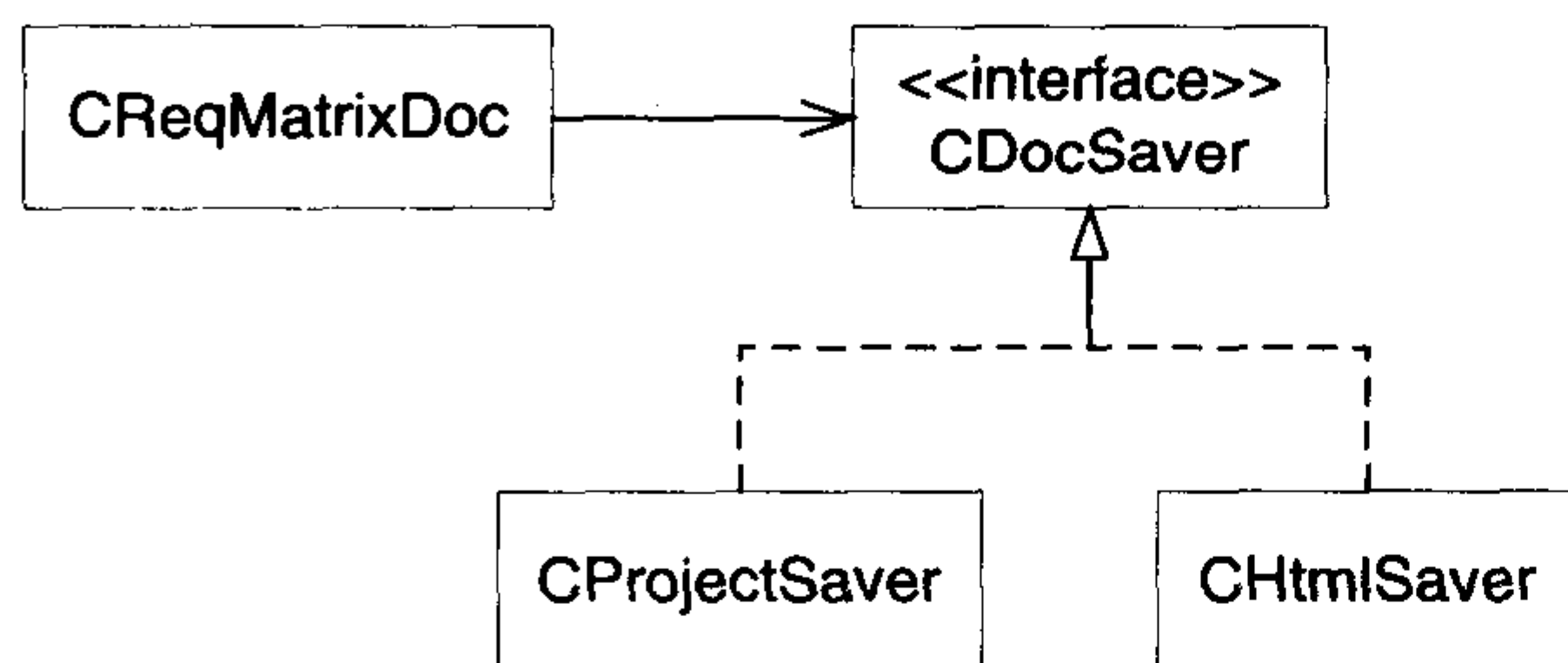


图 24-12 依赖方向与开闭原则

其中，我们运用了依赖倒置原则（Dependency-Inversion Principle）惯用的“用两个抽象依赖代替一个具体依赖”策略，我们引入了一个接口 CDocSaver，然后让 CProjectSaver 实现这个接口。这个设计满足开放封闭原则，究其原因，最关键的一点就是 CProjectSaver 对 CDocSaver 接口的单向依赖。我们只需新写一个 CHtmlSaver 来实现接口 CDocSaver，就可以很快支持“导出为 HTML 格式的网页”特性。

## 24.4 着色顶点与 UML 应用

### 24.4.1 着色顶点

在图论的基本理论中，有加权顶点的概念。顶点具有的与之相关的数，称为权（weight）；该顶点称为加权顶点。

这里我们做个扩充，引入着色顶点的概念：顶点具有的与之相关的颜色，称为着色（color）；该顶点称为着色顶点。

举个例子，讨论图的定义时的图，现在每个顶点都被着了色。着色顶点可以表达更为丰富的含义：比如在图 24-13 中，红色、蓝色和黄色的顶点可以分别代表大、中、小型城市，边代表它们直接的直达航班；从中型城市 V1 到中型城市 V6，没有直达航班，要在大型城市 V4 转乘，一目了然。

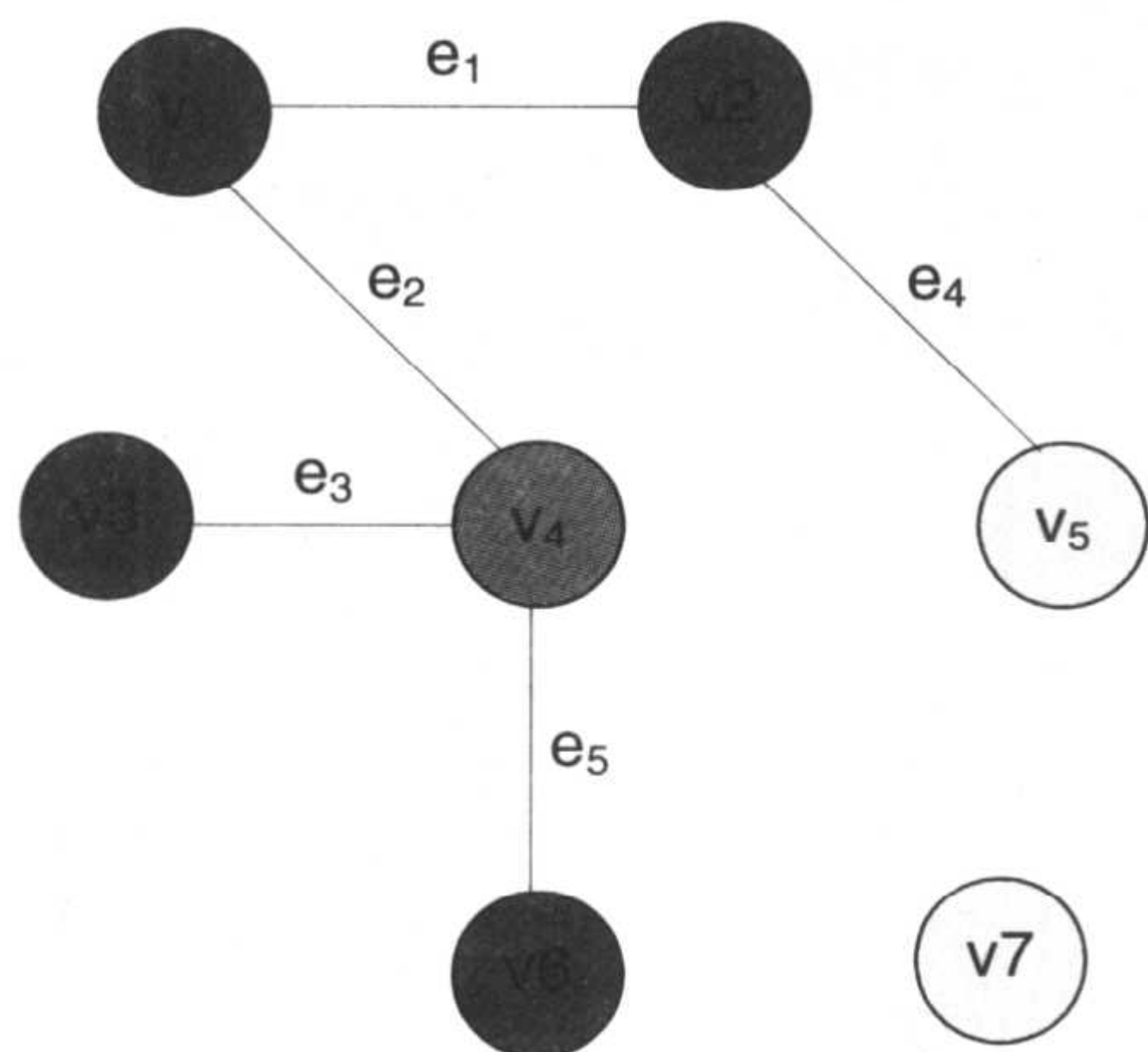


图 24-13 直飞航班图

### 24.4.2 着色顶点的 UML 应用——通过颜色为图元分类

用 UML 为问题及解决方案建模的时候，有时 UML 图会比较复杂，不容易读懂。下面以两个例子来说明如何运用着色顶点的方法，提高 UML 图的易读性。

图 24-14 是 J2SE 集合类框架的 UML 图，运用了蓝、黄两种彩色。每个看这幅 UML 图的人，在留意众多细节之前，都会首先注意到本图的大局——多个蓝色的类组成的层次结构，周围分布着一些黄色的类。继续看下去，蓝色的类和黄色的类之间使用的是“实现”关系——于是知道黄色的类是接口。至此，整个结构清楚了——多个具体类组成了类层次结构，不同的具体类可能实现特定接口。接下来，分析该 UML 图的语义细节……

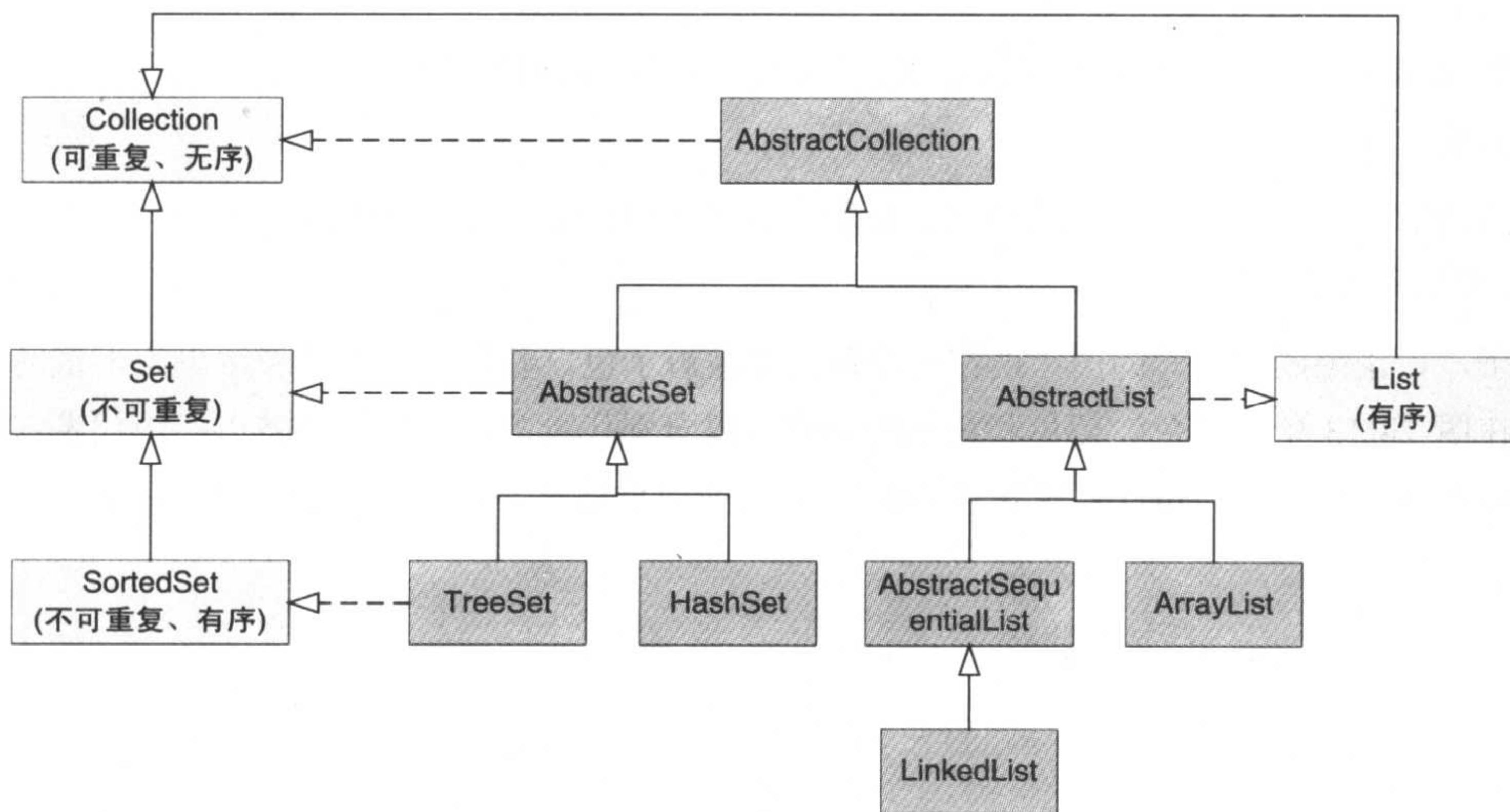


图 24-14 J2SE 集合类



再举一例。本书将在第 26 章的图 26-2 中试图为整个软件工程学科建立概念模型，涉及到的图元比较多，概念也比较复杂，所以使用了黄、红、绿、蓝等多种彩色。实际上，上述四种颜色分别代表过程、项目管理、方法论、任务四个子范畴；一方面，这些色彩的含义要大致读懂了该 UML 图之后，才能反映到读者的脑海中；另一方面，其实在读者看图过程中，这些色彩就起了很大作用了——它们将一幅比较大的类图划分成四个相对独立的小区域，使读者看起图来有条不紊。

建模的本质是抽象，抽象的目的是把握重点，以使结构清晰化。如果我们画的 UML 图太复杂，那就应当想一想，我们是不是犯了“为建模而建模”的毛病，忘记了原本的目的呢？

使 UML 图清晰易读的方法有很多，为图元着色就是其中一种，并且它是令人愉快的。仔细想来，上面两例都是用颜色为图元分类，这种分类带来了两条令人兴奋的好处：其一，为原图附加了一层大局信息，这些大局信息会被读者最先注意到，并为进一步理解图的细节打下基础；其二，由于大局信息是通过着色的方式提供的，并没有引入任何新的图元，所以图的复杂性并没有增加。

### 24.4.3 着色顶点的 UML 应用——UML 彩色建模方法介绍

如果问我，看完《人月神话》这本书，给我印象最深的一个词是什么，我会毫不犹豫地回答：概念完整性。为了达到概念完整性，领域建模（domain modeling）是非常重要的一个环节。可喜的是，领域建模已经越来越被业界所重视，也有不少好书和好的方法浮出水面。本节介绍 Peter Coad 大师的著作《Java Modeling In Color With UML: Enterprise Components and Process》中讲述的彩色建模方法——一种领域建模方面的优秀技术。

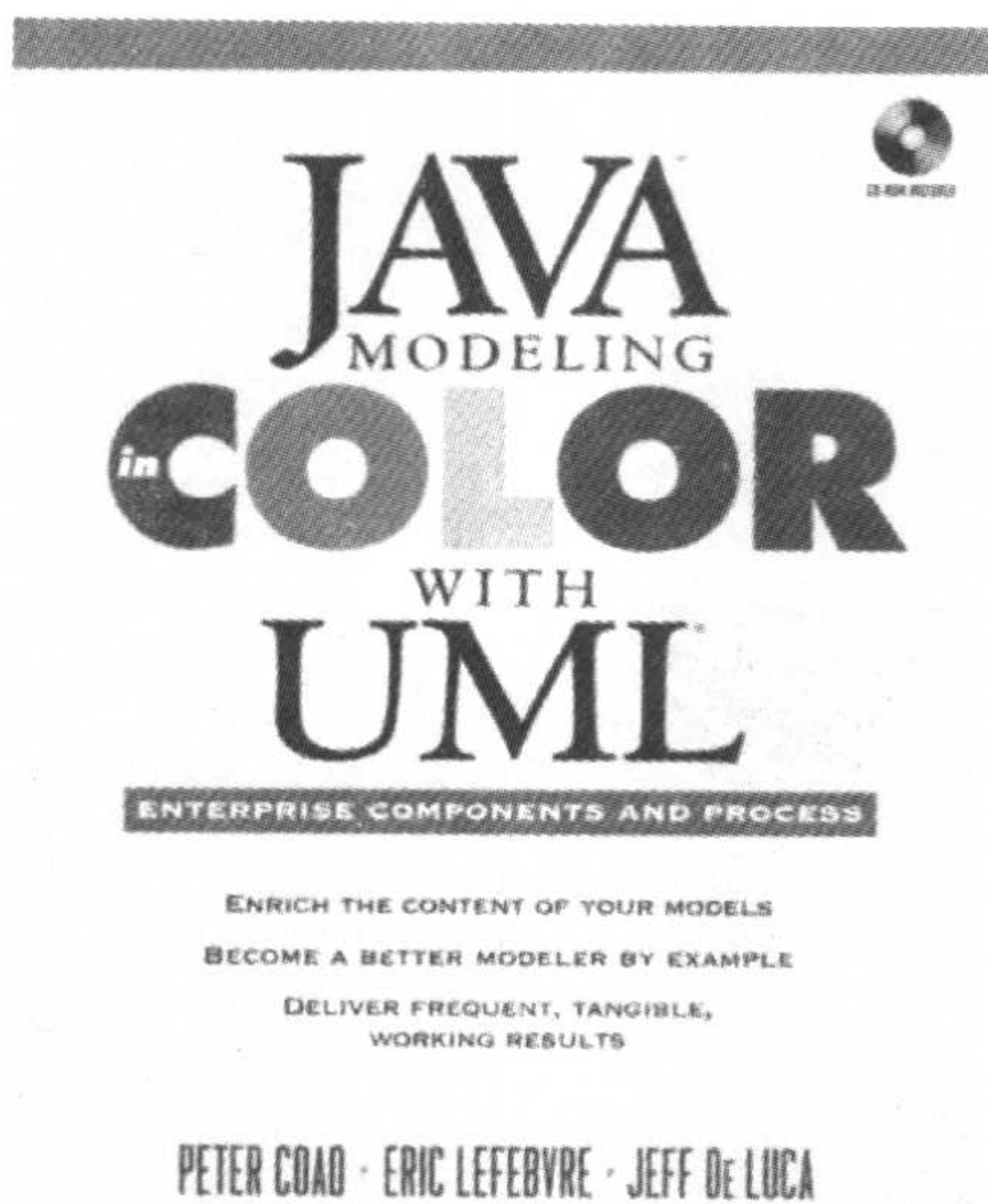


图 24-15 Peter Coad 的“彩色书”



Peter Coad 认为，领域模型主要由四种元素组成，它们分别是：瞬间事件（MomentInterval）、角色（Role）、人—物—地点（PartyPlaceThing）、描述（Description）。彩色建模技术的目的是“为模型增加一层‘视觉可监测的’信息”，具体而言，它定义了四种颜色分别表示上述四种领域元素。

- 粉红：代表“瞬间事件”。在模型中，瞬间事件往往封装了我们最感兴趣的方法（method），这些方法和系统将来的功能有直接的联系，它们可以说是模型的灵魂。它们也是最容易变的，因此选用了最活跃的粉红色代表；
- 黄色：代表“角色”。瞬间事件的发生，往往会牵涉到多个角色，角色具体由“人—物—地点”来扮演。角色意味着在特定场景下的责任，它们也是比较容易变化的，但比起瞬间事件还是要稳定些，所以用比较活泼的黄色代表；
- 绿色：代表“人—物—地点”。业务领域不同，会牵涉到不同的人、物、地点。它们都比较稳定，用安静的绿色表示；
- 蓝色：代表“描述”。最后为人、物、地点引入更抽象一级的描述元素，这些描述元素可以被多个不同的“人—物—地点”所共享。比如 Date 就是描述元素，人可以有生日，汽车可以有生产日期，等等。描述是最为稳定的一类模型元素，用稳定得几乎忧郁的蓝色代表。

例如，图 24-16 是网管软件领域建模时的模型之一角。Ping 是瞬间事件元素，其涉及两个角色元素：Pinger 和 Pingee。这两个角色都由 IPDevice 来扮演，IPDevice 属于“人—物—地点”元素。IPDevice 拥有零到多个 IPAdress，IPAdress 是描述元素。当然，IPDevice 和其四个子类 Router、Switch、Server 和 Host 都是“人—物—地点”元素。

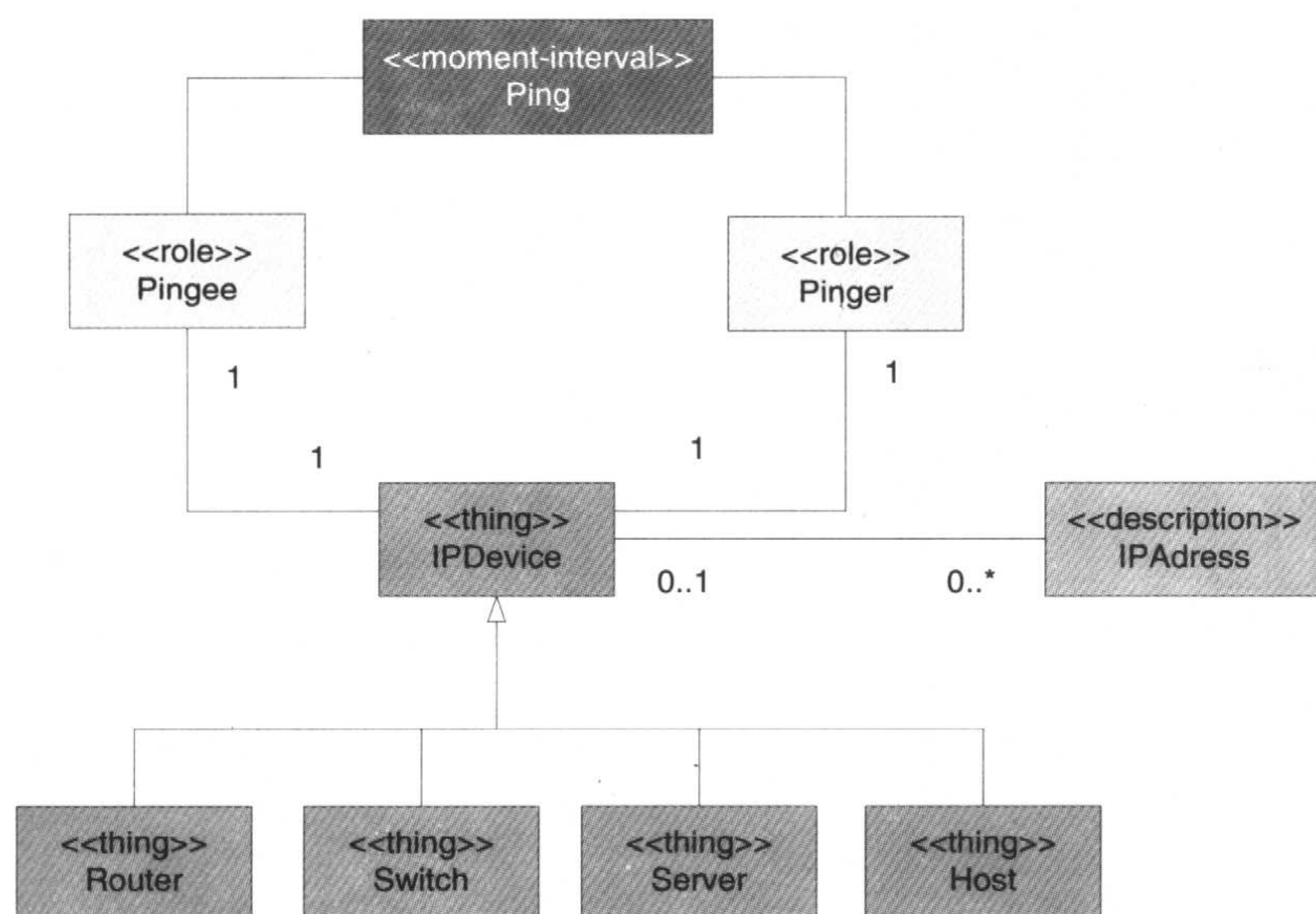


图 24-16 彩色建模一例



Edward R. Tufte 在其经典著作《Envisioning Information》中指出，颜色在信息设计中的基本作用有四：

- 分类
- 度量
- 模仿
- 装饰

彩色建模方法充分利用了颜色的分类和度量功能（当然经过颜色装饰的模型也比原来更赏心悦目）：它不仅利用了四种颜色来为图元分类，而且粉红、黄、绿、蓝四种颜色分别代表不同等级的“易变程度”，具有度量意义。

## 24.5 着色边与 UML 应用

在上文中，我们对经典图论进行了小小扩充，并且体会了这种扩充的应用价值。当然，也可以对边的概念进行类似的扩充——着色边。

合理处理依赖关系，是“拥抱变化”的关键所在。结合第 21 章“敏捷设计：从理论到实践”中所述的良性依赖原则，可以将依赖的良、恶属性可视化，分别用绿色和红色表示。

其实，已经有很多 UML 建模工具借助“着色边”提供更多对设计师有用的信息或使信息更明显以提高工具的易用性。例如，Borland Together，以及 Omondo EclipseUML 等基于 Eclipse 的建模工具，对着色边的应用虽然不同，但都有出彩之处。

## 24.6 图的同构与 UML 应用

### 24.6.1 图的同构

称图 G 与图 H 同构，如果

- (1) 存在  $V(G)$  与  $V(H)$  的一一对应
- (2) 存在  $E(G)$  与  $E(H)$  的一一对应
- (3) 在(1)与(2)的一一对应保持顶点与边的关联关系不变

比如图 24-17 所示的两个 UML 类图，画法虽各异，但其语义是完全相同的。

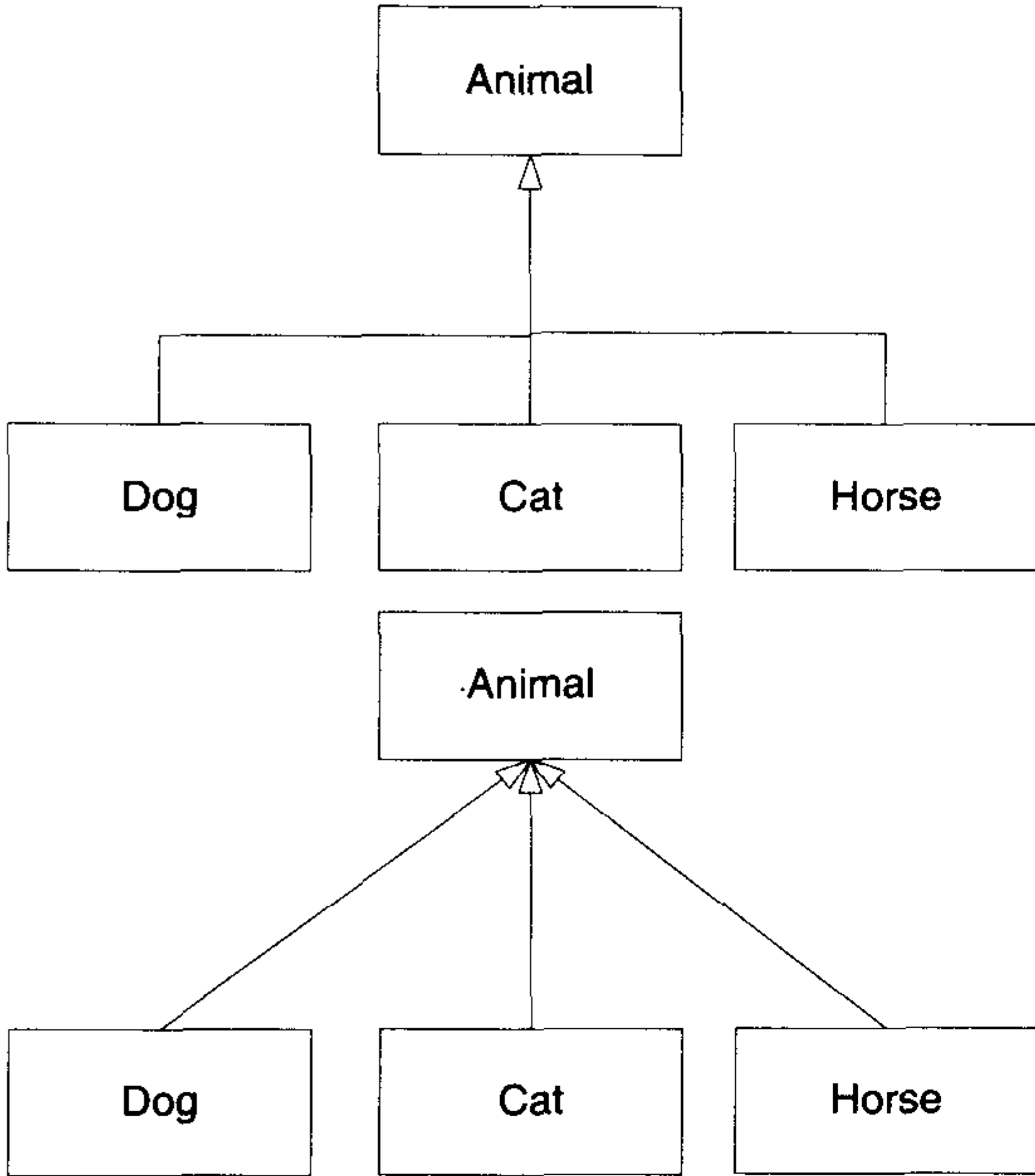


图 24-17 画法不同，语义相同

24.6.2 图的同构的 UML 应用——UML 风格

互为同构的 UML 图，其语义虽然是完全相同的，但其易读性，却可能相差极大。因此，我们提倡好的 UML 风格。

图 24-18 是一幅 UML 状态图，描述了 Java 线程的状态变化情况。该图的起始点在最左边，终止点在最右边，始态和终态分别在次左边和次右边，整个图结构清晰，语义一目了然。

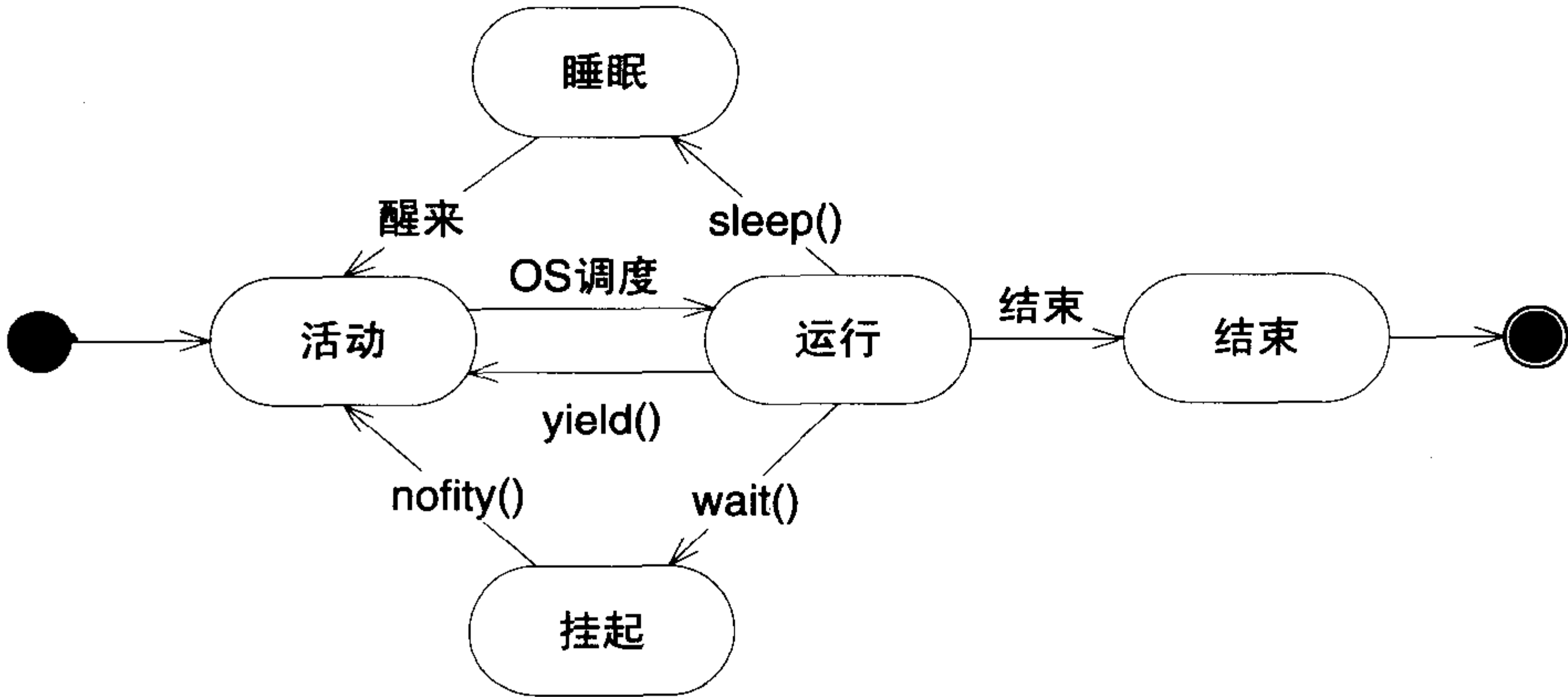


图 24-18 布局合理的状态图

再看这幅图 24-19，语义完全相同，但是看起来却费劲多了。

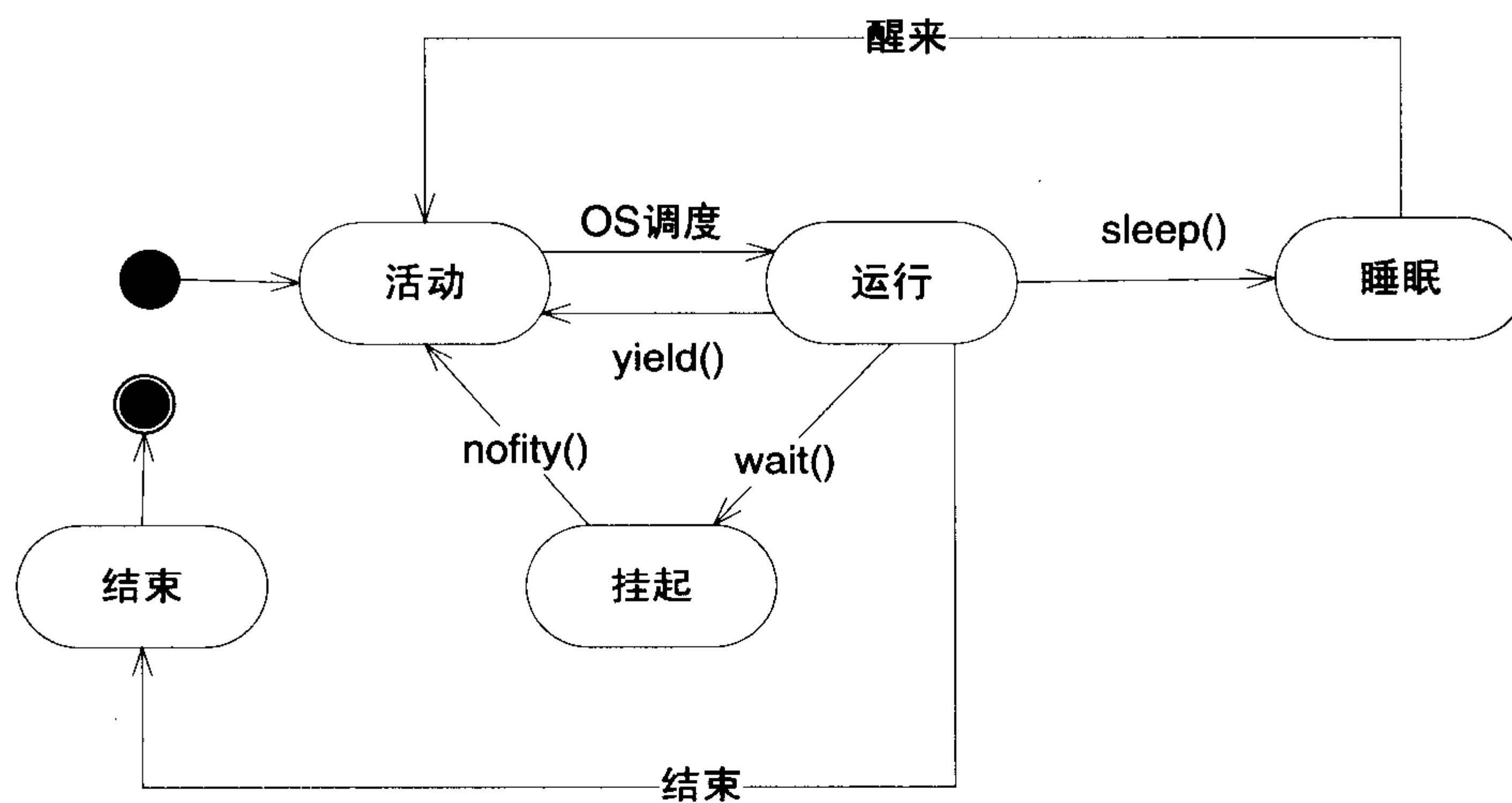


图 24-19 布局不合理的状态图

这里不再举更多例子了，《UML 风格》（The Elements of UML Style）一书，汇集了很多绘制 UML 图的专家级建议，推荐一读。





## 第 25 章 理解软件过程：解析 RUP 核心概念

---

过程是方法的自然延续。

——Ivar Jacobson, 《统一软件开发过程之路》

随着程序员渐渐成长为软件架构师，他的职责面越来越广。简而言之，在这个过程中有一个从只关心技术到技术和管理并重的转变。

软件过程规定了团队协作的方式，乃是软件架构师所必须了解的。作为著名的软件过程之一，RUP (IBM Rational Unified Process, 统一软件过程) 的好处是影响广泛、资料丰富。在实践中，笔者发现，对 RUP (以及一般意义上的软件过程) 的重要概念理解不到位，特别是对概念之间的关系理解不到位，是阻碍人们成功应用 RUP 的主要瓶颈之一。

本章要讨论的主题和 RUP 有关，但讨论内容并不局限于 RUP:

- 既有和 RUP 相关的
- 也有超出任何具体软件过程的通用概念
- 还有软件架构师的工作职责等现实问题

### 25.1 架构师必须了解软件过程

---

#### 25.1.1 架构师的工作职责

作为软件架构师，也应当了解软件过程吗？回答是肯定的。

并且，软件架构师了解软件过程是非常重要的，这和软件架构师要承担多项职责有关。一般而言，软件架构师应担负如下职责：

- 领导并负责架构设计
- 实际参与架构原型的开发实现
- 讲解架构，指导开发，协调冲突
- 为项目管理提供支持，如技术可行性、任务划分、人员招聘等

- 了解所在组织的业务目标，令架构更好地支持业务目标
- 评估新技术并提出采用建议

### 25.1.2 架构师必须了解软件过程

正如上面所列出的工作职责所暗示的那样，软件架构师要和多个部门或小组打交道：例如，他本人可能是架构设计组的领导；另外，他还要指导不同程序团队以架构为中心进行开发，并协调他们的合作，解决他们之间的冲突；软件架构师还要支持项目经理的工作；为了深入洞察组织的业务目标，架构师还要和市场部门打交道，甚至要拜访客户……因此，架构师必须有开阔的“视野”，可以纵观整个软件过程的全局，并对不同角色相互合作的接口和时机有清晰的把握。

一句话，软件过程规定了打交道的接口，软件架构师需要了解它。

## 25.2 RUP 实践中的常见问题

RUP 著名的二维结构，其时间维相关的概念有阶段、迭代、里程碑等，内容维相关概念有 workflow、角色、活动、工件等。但不少人对这些概念理解不深，特别是对概念之间的关系把握不到位，造成在实践中出现问题。

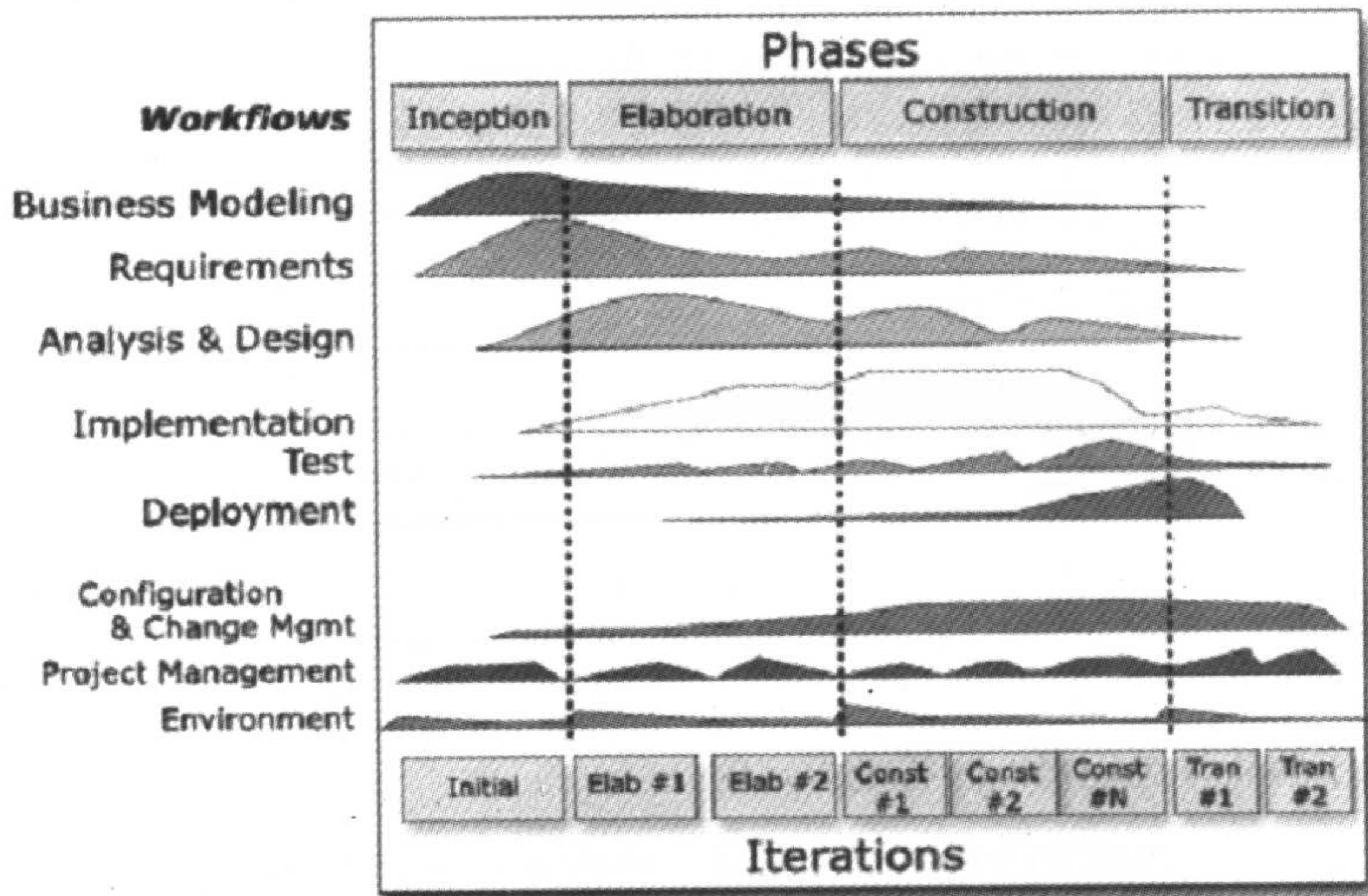


图 25-1 RUP 的二维结构（图片来源：RUP）

另外，就是迭代式开发——这种包括 RUP 在内的多种软件工程过程都一致推崇的最佳实践——和活动、工件这些基本概念有何关系。不知道迭代和活动、工件的关系，实际应用 RUP 时又如何贯彻迭代式开发的思想呢？

还有，配置和变更管理对所有现代软件开发过程都是必不可少的支持活动，RUP 更是将其列为“RUP 的 6 大最佳实践”之一。但笔者发现，不少开发人员认为配置和变更管理太麻烦，仅仅是因为他们没有理解配置和变更管理和工件的基本关系。

我们的任务，就是要解决这些问题。

## 25.3 RUP 核心概念解析

下面采用“为概念及其关系建模”的方法，对概念及其关系进行考察，以期深入理解 RUP 的核心概念。

### 25.3.1 一图胜千言

图 25-2 是一幅 UML 类图，它概括了上述问题的相关概念，并着重表达了概念之间的关系。本图的丰富语义，我们通过下面几节详细来分析。

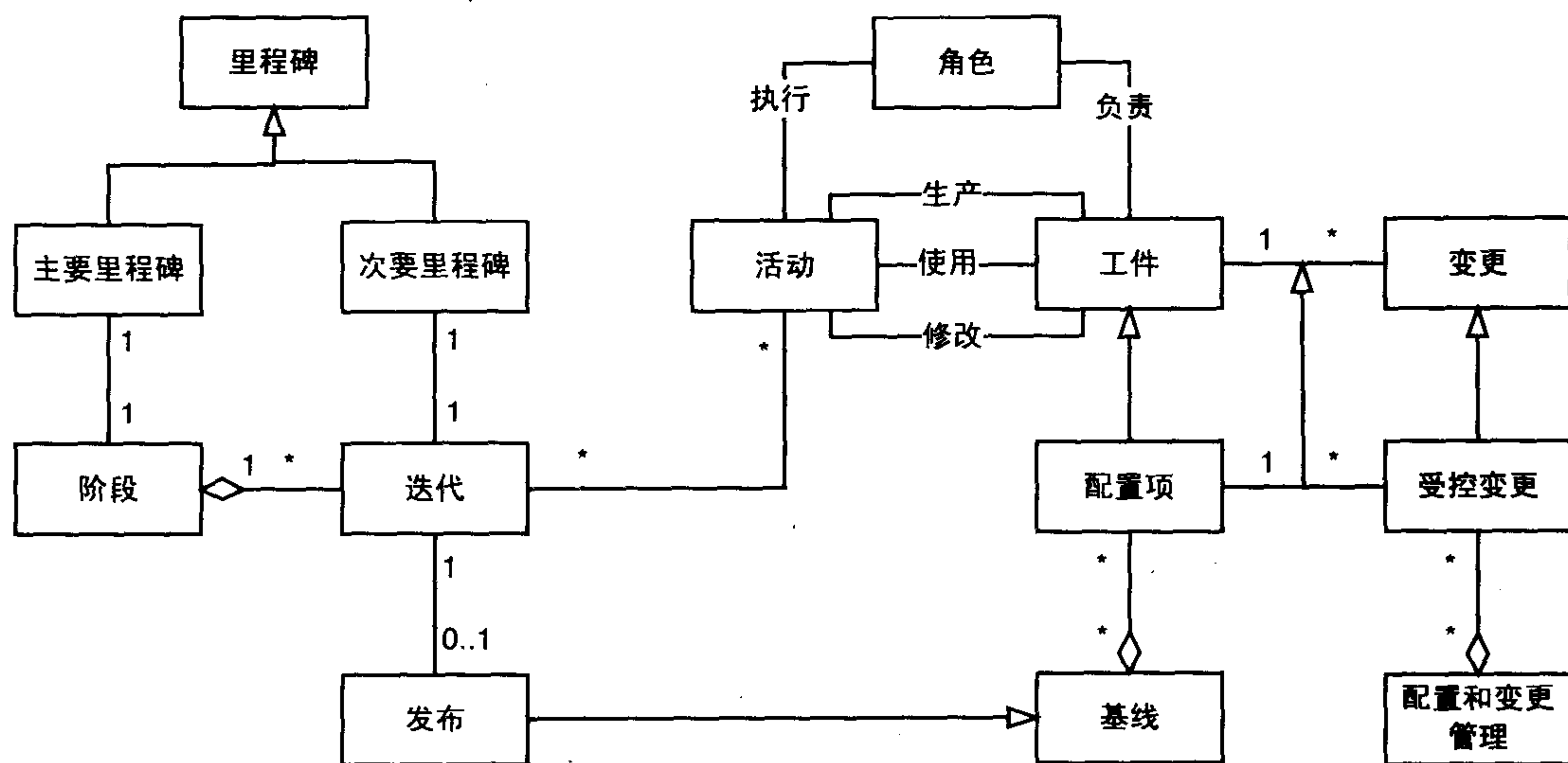


图 25-2 RUP 核心概念及其关系

### 25.3.2 角色执行活动，活动生产工件

任何软件工程过程，都少不了角色 (role)、活动 (activity)、工件 (artifact) 等概念 (或者类似概念)。如图 25-3 所示。



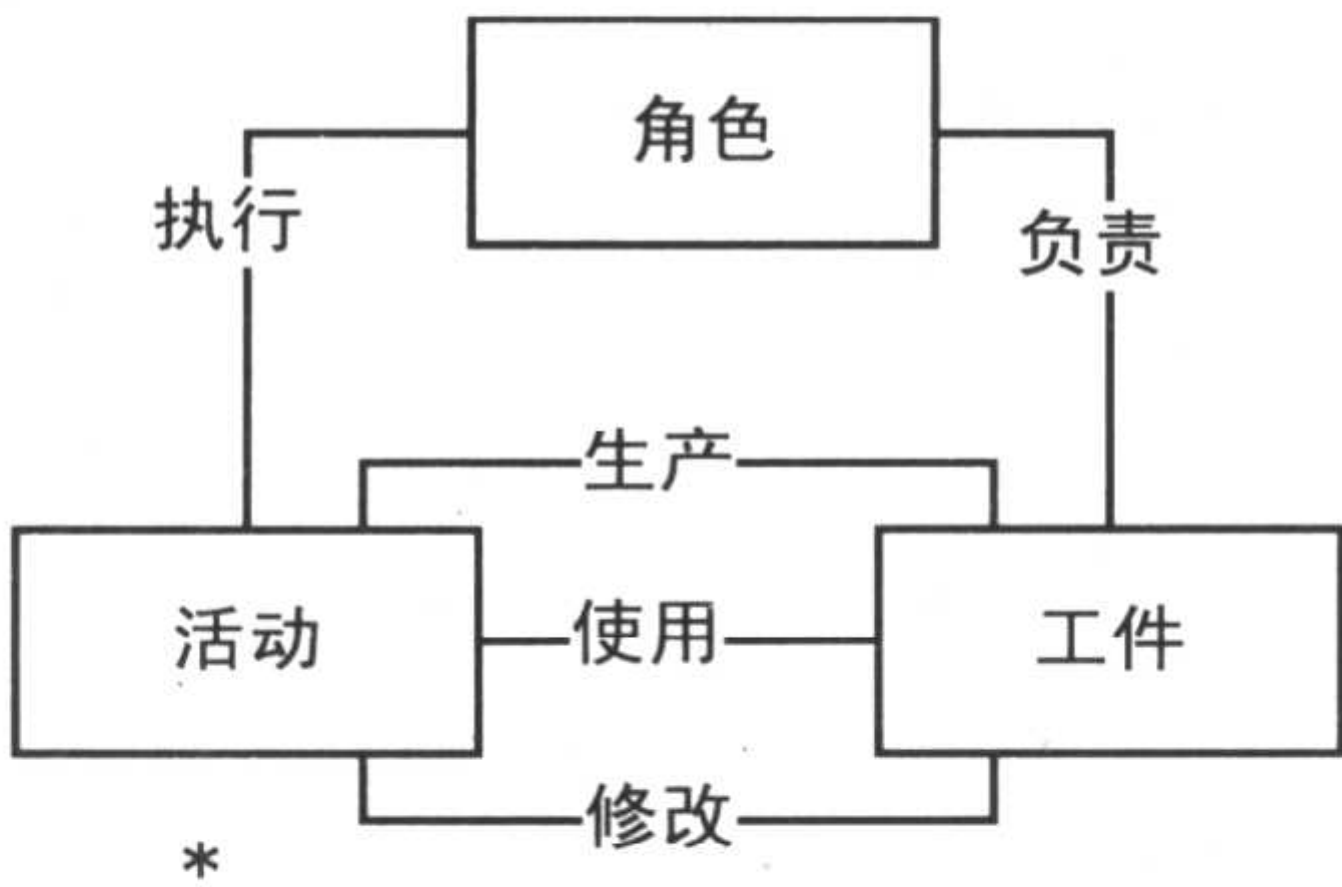


图 25-3 角色、活动和工作件

这些概念本身很好理解。角色是对个人或者作为开发团队的一组人的职责的规定；具体人和角色的关系，好比人和帽子的关系。活动就是角色执行的工作单元。工作件就是工作的成品或半成品。

倒是这些概念的关系显得更加重要，如图 25-4 所示。角色的职责，具体体现在他执行活动和负责工作件上。工作件是由活动生产出来的——工作件是活动的输出；比如制定《编码规范》。然而，活动本身也可能以工作件为输入——活动可能要求使用工作件；比如编码活动要参考《编码规范》。还有一种关系，工作件既是活动的输入又是它的输出——活动修改工作件；比如修改《编码规范》。

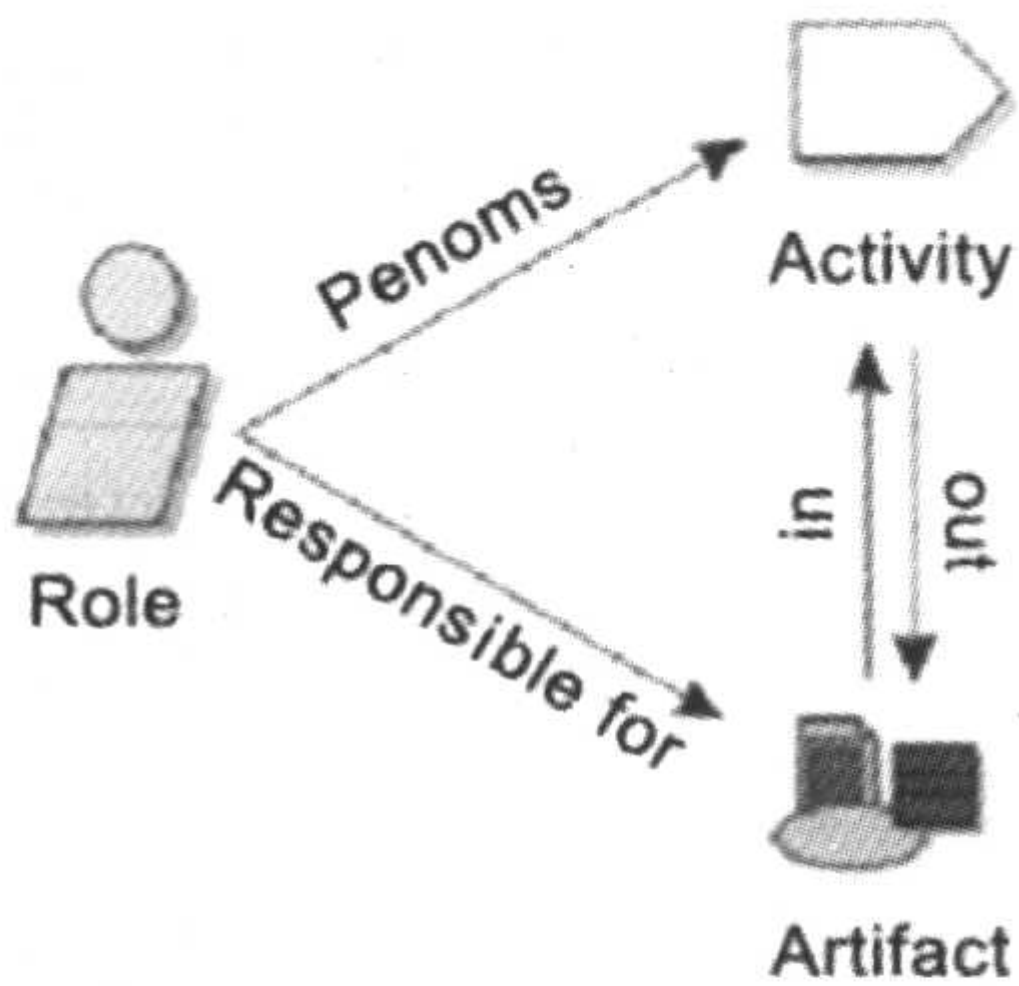


图 25-4 角色、活动和工作件之间的关系（图片来源：RUP）

### 25.3.3 阶段和迭代：提供不同级别的决策时机

尽早解决重大风险，是软件工程管理中的一条重要原则。正如 Tom Glib 所说：“如果我们不主动化解风险，那么它们会自己找上门来。”心存侥幸是很危险的。

RUP 是风险驱动的。它将整个开发生命周期分为 4 个阶段：初始阶段、细化阶段、构造阶段、移交阶段（如图 25-5 所示）。初始阶段着重化解业务风险，并确保所有涉众对项目达成一致



的认识。细化阶段主要化解技术风险，要定义并创建可执行的系统架构。相对而言，当决定开始构造阶段的时候，风险已经比较小了。当然，风险是动态变化的，构造阶段和移交阶段照样会有这样那样的风险存在。

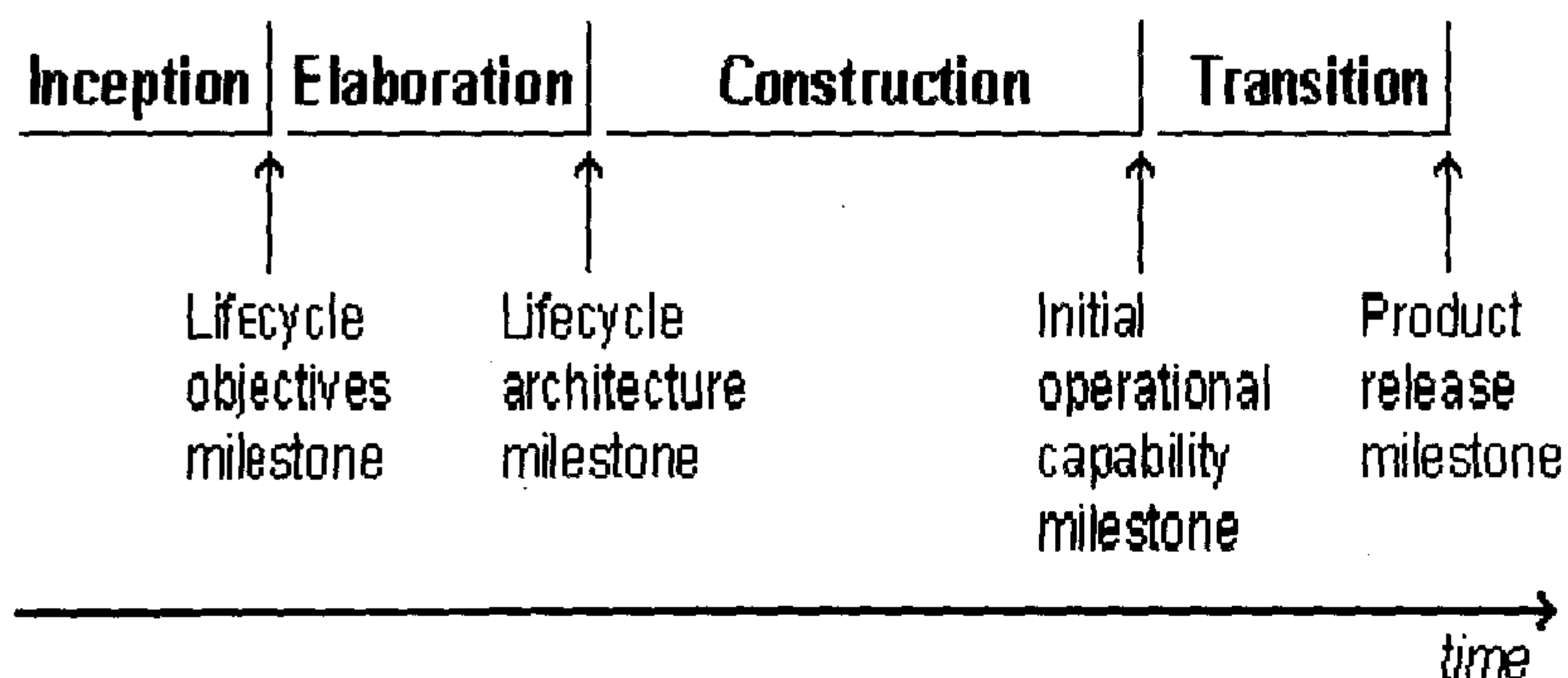


图 25-5 RUP 阶段（图片来源：RUP）

RUP 的每个阶段又可分为一到多个迭代周期。采用迭代式开发，意味着有持续不断的反馈（如图 25-6 所示）；反馈是决策的基础，也是化解风险的基础。迭代式开发的一个主要目的就是尽早降低风险，通过每次迭代中分析、按重要性排序并解决主要风险，来达到尽早化解风险的目的。

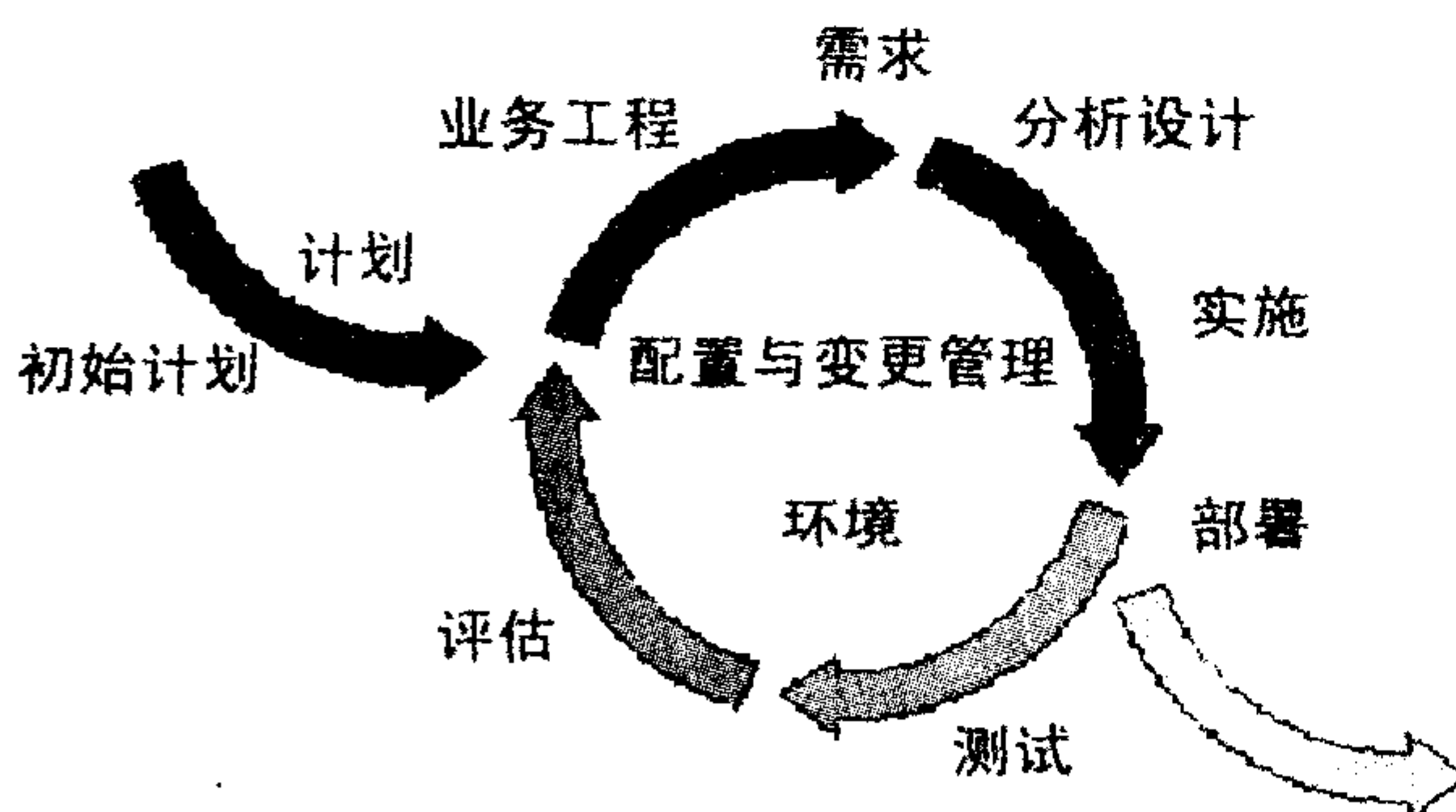


图 25-6 RUP 采用迭代式开发（图片来源：RUP）

这个根据持续反馈来进行决策的时机，叫做里程碑（Milestone）。里程碑有两种，阶段结束对应的里程碑叫做主要里程碑（Major milestone）；迭代结束对应的里程碑叫做次要里程碑（Minor milestone）。可以说，阶段和迭代，为开发团队提供了不同级别的决策时机。如图 25-7 所示。

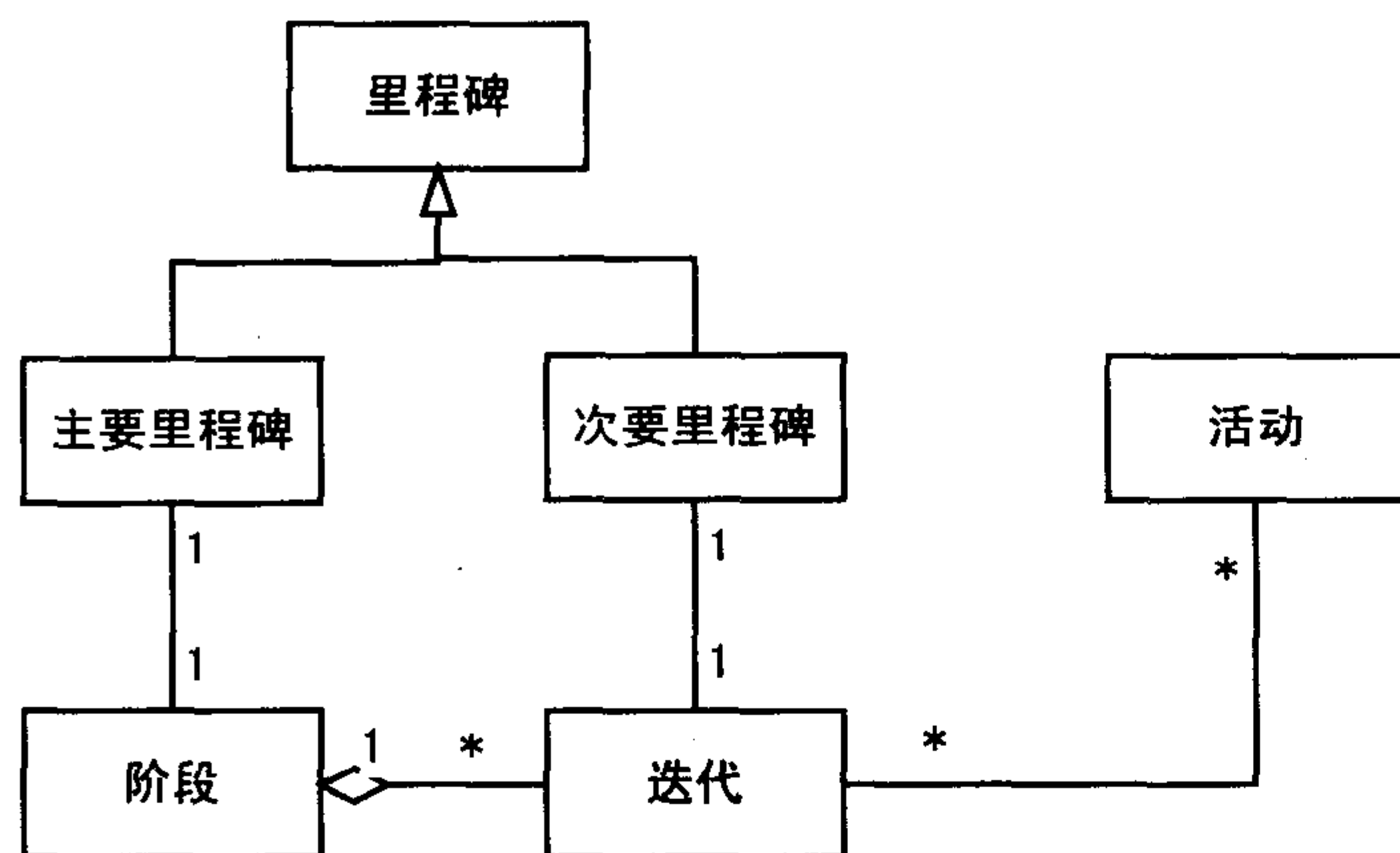


图 25-7 RUP 中的两种里程碑

上图清晰的表达了上述分析的思想。里程碑分为主要里程碑和次要里程碑两种；阶段可以包含多次迭代；每个阶段必然有一个主要里程碑标识结束，每个迭代必然以一个次要里程碑标识结束。

迭代的具体进行，是要靠角色执行相关活动。上图中的迭代和活动的多对多关系，精彩地反映了迭代开发的特点——每次迭代都执行多个（甚至全部）开发活动。

### 25.3.4 配置和变更管理支持迭代式的基于基线的开发

迭代式开发意味着每个后继迭代，都以前一个迭代为基础，不断地进化和完善系统，直至完成最终产品。历史上有一段时期，为了强调这一点，RUP 曾特意宣传“迭代和增量开发”。

建立并管理基线（baseline），为迭代式开发提供了有力支持。基线的要点有二：一是要通过评审，二是要受配置和变更管理控制。IEEE 对于基线的完整定义是：已经通过正式复审和批准的某规约或产品，它因此可以作为进一步开发的基础，并且只能通过正式的变更控制过程进行改变。

配置和变更管理完成建立并管理基线的任务。置于配置和变更管理之下的工件，称为配置项（configuration item）——因此如图 25-8 所示把配置项建模为工件的子类，而基线就是有多个配置项组成的瞬时快照——因为受配置和变更管理控制是基线的必要条件。

上面讨论了“工件—配置项—基线”这些静态概念，下面再讨论一下相关动态概念。任何工件，都有可能发生变更；正如并不是所有工件都是配置项一样，变更也不一定都受控，比如，用于讨论的临时设计就不必受控。只有配置项的变更，才是需要受配置和变更管理控制的。到底哪些工件应当受控，是根据实践情况决定的，应当在规范性和灵活性之间权衡考虑。

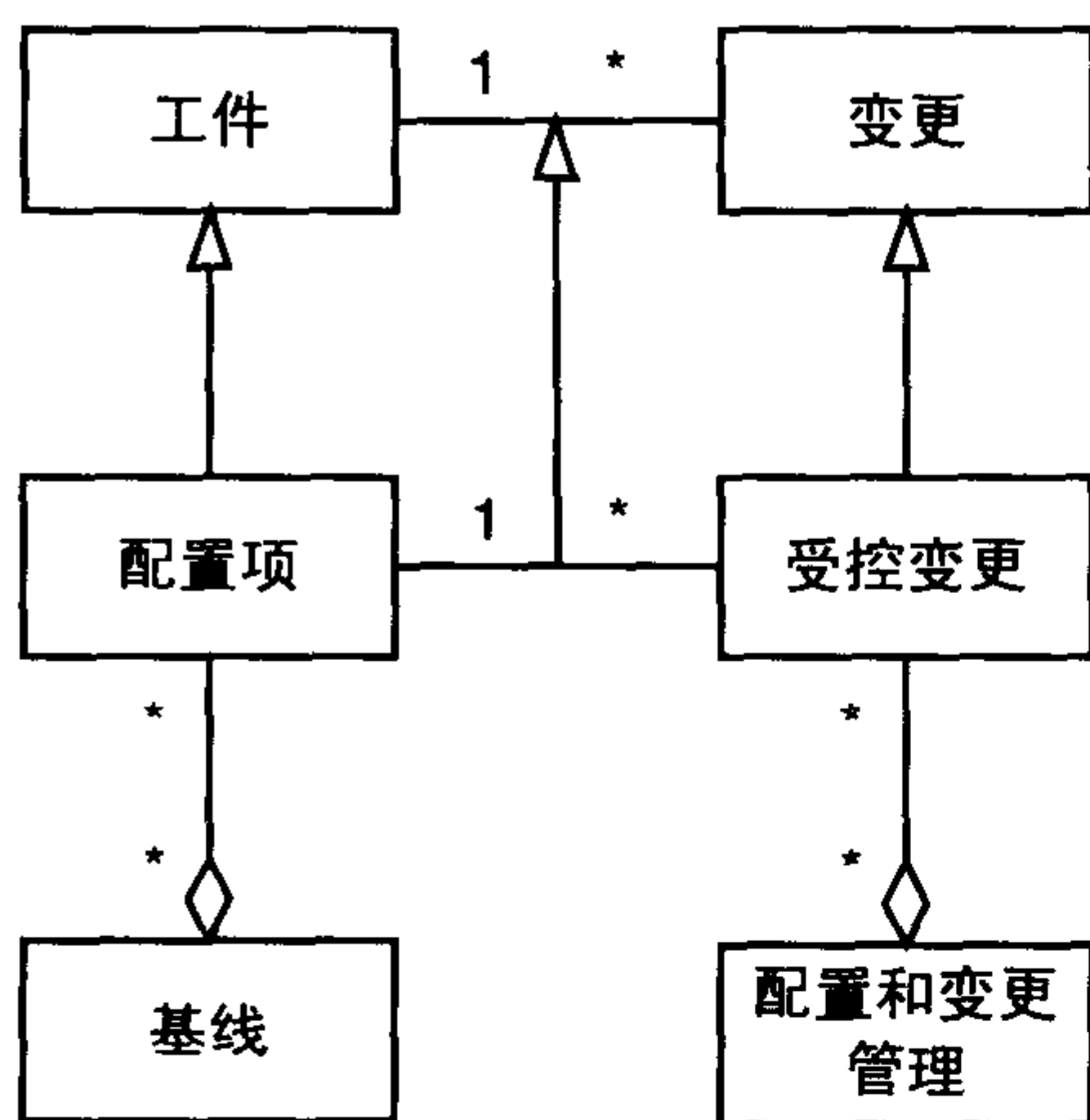


图 25-8 配置管理相关的核心概念

### 25.3.5 发布是什么，发布不是什么

发布（release）是软件产品的一个稳定的、可执行的版本，它包括了发布说明、用户手册等相关工件。

单纯的文档或者不能执行的软件版本，并不是发布。

发布是某个迭代周期的成果，但是并非每个迭代周期都用发布。

图 25-9 中表达得很明白，发布是特殊的基线。这意味着发布不是单一的工件，而是工件集；而且发布的工件都应当置于配置管理的控制之下，这是因为你必须标识和跟踪这些工件，开发团队或者用户的很多活动都和它们相关。



图 25-9 发布





## 第 26 章 海阔凭鱼跃：通盘理解软件工程

---

模型可以澄清相互间的关系，识别出关键元素，有意识地减少可能引起的混淆。

——凯文·福斯伯格，《可视化项目管理》

知识面广的人学习新东西最快。

软件架构师的知识面必须广，因为工作性质决定了架构师必须保持对各种技术的“开放”。作为软件架构师，你不仅应该能够快速掌握软件工程新技术，还应该能够对众多技术进行综合运用，以取得最佳的实践效果。这是很现实的问题。

本书认为，通盘把握软件工程概念模型，将对解决上述问题大有裨益。

### 26.1 什么是软件工程概念模型

---

模型就是抽象，就是有意识地忽略事物的某些特征。抽象带来的好处是能够反映模型中元素之间的关系，清晰把握大局。

概念模型是模型的一种，简单说就是抽象程度极高的一种模型。

软件工程概念模型是对软件工程领域进行抽象描述的模型，它能够使我们对软件工程有一个完整把握。

### 26.2 一个精简的软件工程概念模型

---

《软件工程——技术、方法与环境》一书中，有一个极为精简的软件工程概念模型：

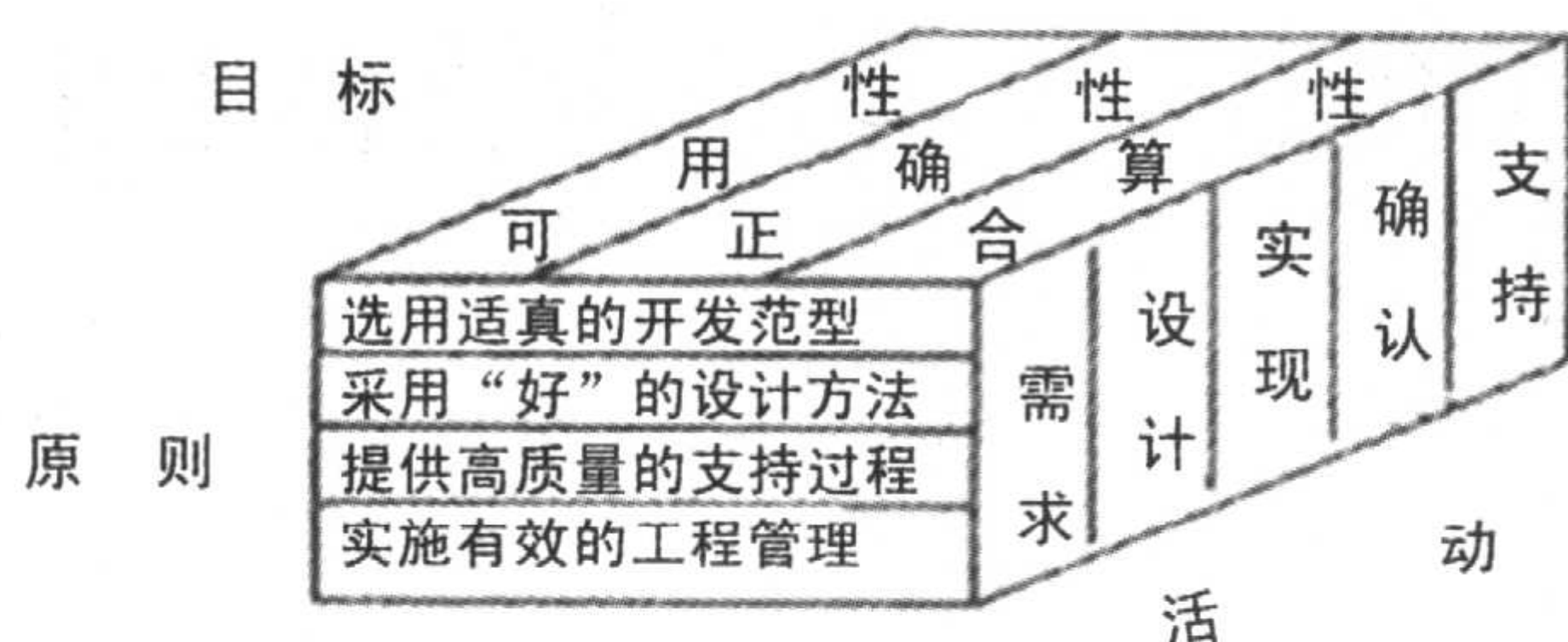


图 26-1 软件工程概念模型（图片来源：《软件工程——技术、方法与环境》）

该模型可以用一句话概括：软件工程是（目标，方法，活动）三元组。它体现了“目标—方法—活动”的 3 维正交关系：

- 任何目标，都要依照特定方法，由特定活动实现
- 任何方法，都是指导特定活动，来完成某种目标
- 任何活动，都由特定方法指导，来完成某种目标

## 26.3 一个细化的软件工程概念模型

图 26-2 是笔者理解的软件工程概念模型（采用 UML 类图的语法）：

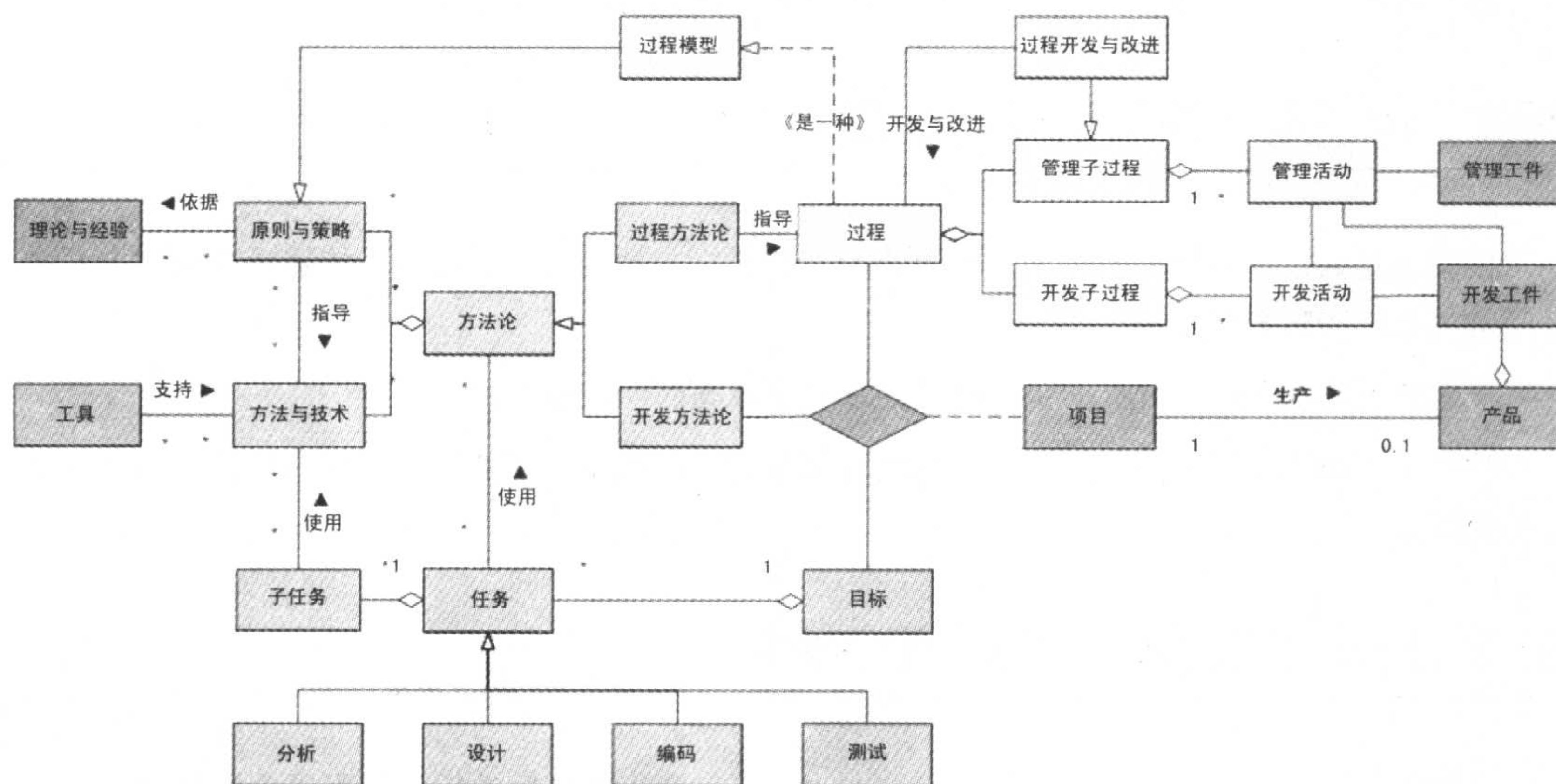


图 26-2 软件工程概念模型

### 26.3.1 模型概述

图中，“理论与经验”和“工具”可以认为是两个比较独立的概念，其他概念可以被分为 4 组——“方法论”、“过程”、“目标”、“项目”，分别标以不同颜色。这 4 组主要概念构成了软件工程概念模型的骨架，可以描述为：为达到一定的“目标”，我们建立起相应的“项目”，在某种“方法论”的指导下，按照一定的“过程”，生产出相应的软件“产品”。

从这个模型的骨架中，我们能清晰看到上面精简模型的影子——（目标，方法，活动）三元组。但显著区别是，更加强调“活动”的组织和控制方式——“过程”。这是软件实践发展的必然结果，因为，随着软件产品的复杂程度不断提高，势必要更加强调“过程”。

### 26.3.2 方法论

“方法论”是在一定“原则与策略”指导下的一套相关的“方法与技术”，而“方法论”可以分为“开发方法论”和“过程方法论”两种。相应的，“原则与策略”可以是开发策略，例如著名的“功能分解”策略；也可以是过程策略，例如迭代模型等“过程模型”，就是过程策略。

应当说，“过程方法论”是随着软件实践的深入，在“开发方法论”产生之后才产生的概念。Roger S. Pressman 在其经典著作《软件工程：实践者的研究方法》里就指出：大约每隔 5 至 10 年，软件界就会重定义“问题”，将其焦点从产品转移到过程。在本章后面的部分，笔者将用“过程方法论”的概念解释“Agile 到底是过程还是方法论”的迷惑。

另外，值得一提的是，在实际当中，存在“方法”其实是指“方法论”的现象，在此说明一下（如图 26-3 所示）。一方面，“方法论”是为完成特定目的一套“方法”，“方法论”和“方法”是一对多的关系；另一方面，实际中人们常将“方法论”简称为“方法”，所以也可以认为“方法”是个递归的概念，它可以是“原子方法”，也可以是“方法论”。至此，当你同时面对“Agile 方法”和“Agile 方法论”这两种说法时，就不必迷惑了。

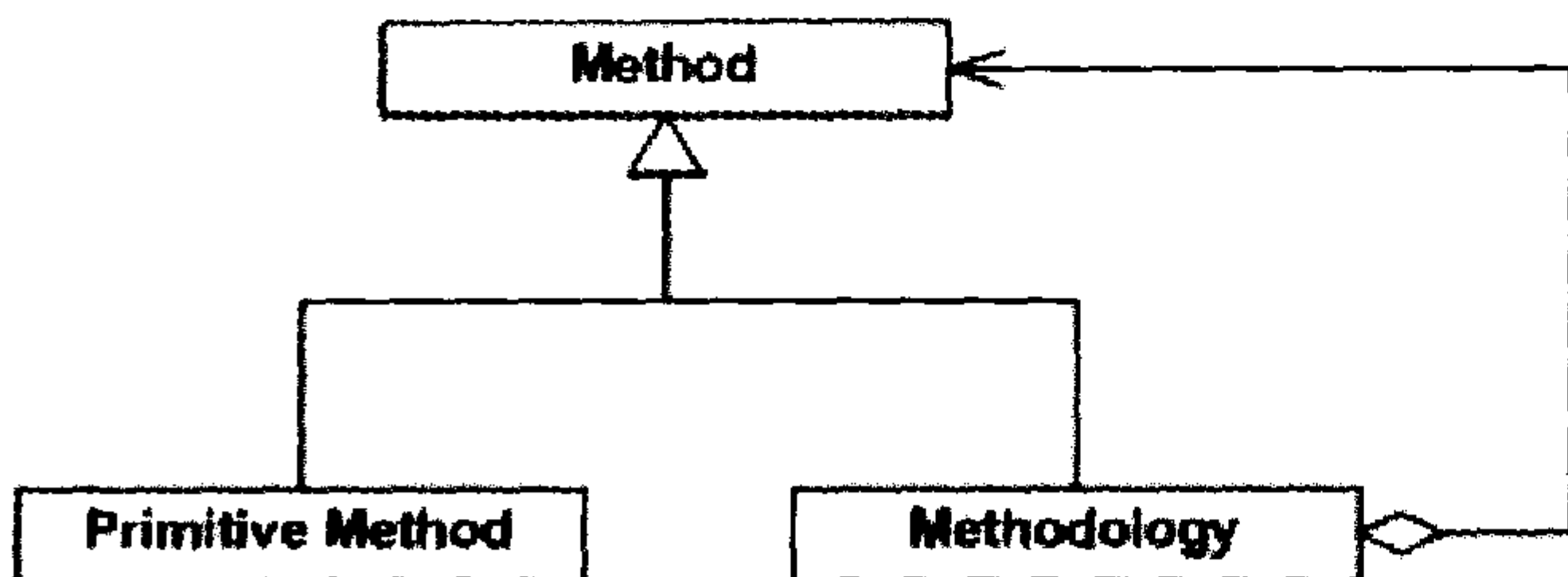


图 26-3 方法论只不过是成套的方法而已

### 26.3.3 过程

“过程模型”是对具体“过程”的抽象，它仅规定了后者的框架，例如瀑布模型规定了“过



程”是线性框架；“过程模型”的例子还有螺旋模型、喷泉模型、迭代模型等。一个具体“过程”包括“开发子过程”和“管理子过程”两个子过程，它们分别由一组相关的“开发活动”和“管理活动”组成。“开发活动”开发出或再加工“开发工件”，“管理活动”使用“管理工件”，对“开发活动”和“开发工件”进行管理。

“过程开发与改进”是一个特殊的“管理子过程”。“过程开发与改进”与其他一般的“管理子过程”相比，后者是为软件产品的开发服务的，而前者是以开发和改进软件过程本身为目的。软件工程大师 Osterweil 在其论文《Software Processes are Software Too》中高屋建瓴地指出：软件过程也是软件。软件有一个开发的过程，软件过程也有一个开发的过程；软件开发产出软件产品，软件过程开发产出过程产品。

RUP 是著名的软件过程产品。CMM 是著名的软件过程改进框架，它本身不是特定软件过程的定义，它只是建议如何一步一个台阶地改进软件过程。在本章后面的部分，笔者将从“过程开发与改进”的角度，谈谈 RUP 的定制和 CMM 的定位问题。

### 26.3.4 目标

任何实践都是有“目标”的，软件实践也不例外；比如“开发 Bug 跟踪系统”就是一个“目标”的例子。“目标”的完成，依赖于一系列“任务”的完成；比如上述“目标”可以分解为“分析”、“设计”、“编码”、“实现”等“任务”。

“任务”通常在“方法论”的指导下完成；比如“分析”任务，可以选用“OOA 方法论”，也可以选用“结构化分析”方法论。每个“任务”还可以进一步分解成多个“子任务”；比如“分析”可以分解为“需求采集”、“需求分析”、“建立分析模型”等“子任务”。

“子任务”通常使用相关“方法与技术”来实现；比如“建立分析模型”子任务，可以选用“UML 建模技术”，也可以选用“结构化建模技术”。

### 26.3.5 项目

“项目”是“目标”、“过程”和“方法论”三者的关联类；这意味着任何一个“项目”，都采用一定的“过程”，在某种“方法论”的指导下，完成某种“目标”。“项目”的“目标”常常就是“软件产品”本身，但也可以不产出任何“软件产品”。“项目”是为完成特定“目标”所做出的临时性努力，可以是建造一栋大楼，一座工厂，也可以是解决某个研究课题，不一而足。

“产品”就是一组“开发工件”的集合，就象经典的“软件”的定义中说的，“软件是程序、文档和相关数据的统称”。“管理工件”是伴随“项目”的进行产生的，但它并不是要交付给最终用户的。



### 26.3.6 其他

现在回过头来看“理论与经验”和“工具”这两个概念。“理论”是高度系统化的知识，“经验”是尚未进行系统化抽象的知识。“理论与经验”是“方法论”的依据，正是在“理论与经验”的指导下，人们总结出方法论的“原则与策略”，又在后者的指导下，人们将众多“方法与技术”组织成一套完整的“方法论”。“工具”用来支持“方法与技术”的，好的“工具”可以提高人们的工作效率，减小出错几率。

其实图中还包含了其他一些信息。比如，“方法”和“工具”是多对多关系：一种“方法”可以采用多种“工具”（比如画 Use Case 图可以采用 Visio、Rose 等多种工具），一种“工具”也可支持多种“方法”（比如 Visio 可以画流程图、UML 图等多种图）。

又比如，“方法”和“过程”是多对多关系：一种“方法”可以被多种“过程”采用（比如 CRC 卡方法可以被 RUP、XP 等多种过程采用），一种“过程”也可采用多种“方法”（比如 RUP 可以采用 OO 建模、结构化建模等多种方法）。

篇幅所限，在此不一一列举了。

## 26.4 软件工程概念模型的具体应用

下面再举几个具体的例子，以说明软件工程概念模型在快速学习、正确理解和深入掌握软件工程技术方面的作用。

### 26.4.1 搞清楚 Agile 是过程还是方法论

当前，“Agile 过程”和“Agile 方法论”的说法都很流行，令初学者相当迷惑。下面根据软件工程概念模型的知识，来弄清这个问题。

好的开始是成功的一半，我要做的第一步就是先推翻“Agile 是过程还是方法论”这个问法，改问“Agile 是过程、开发方法论还是过程方法论”。

第二步，分析《Agile Software Development》一书中给出的 Agile 模型，如图 26-4 所示。

通过对模型的分析，可以看出它是对过程建模：

- 如果去掉人和团队的因素，上面的模型最主要的要素就是过程（Process）、活动（Activities）、产品（Products）和技术（Techniques）了，这显然是个过程的模型；
- 上面的模型中有相当多的和人相关的要素，包括角色（Roles）、技能（Skills）、个性（Personality）、团队（Teams），对人的因素的极其重视，正是 Agile 过程的显著特点；

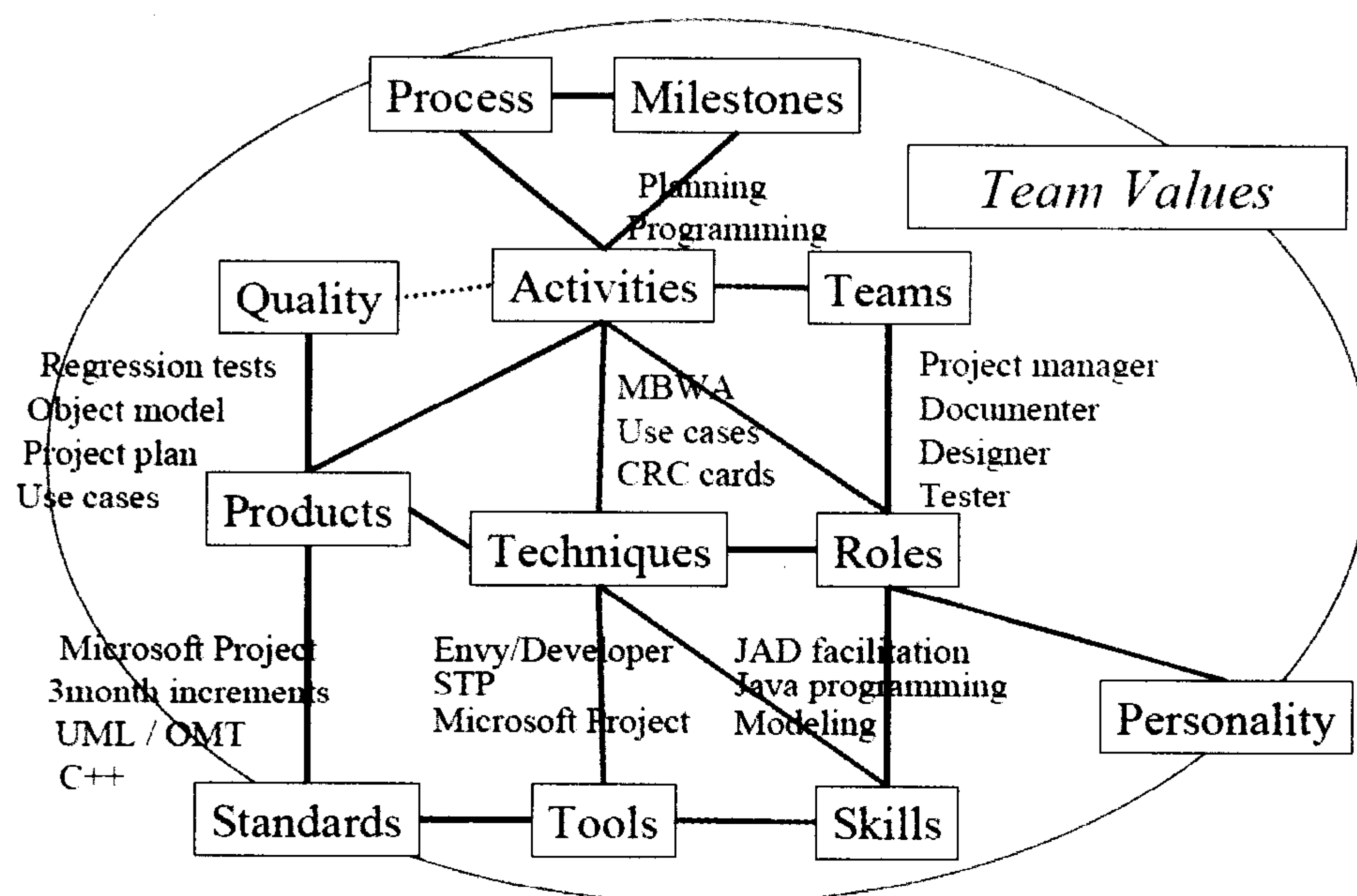


图 26-4 Agile 方法论所涉及的元素（图片来源：《Agile Software Development》）

- Agile 过程是面向人的（people-oriented）过程，实施 Agile 过程的一个关键之处是让人“接受”一个过程而非“强加”一个过程。

好了，我准备提前下结论了——谈过程哪能不谈它背后的方法论呢——Agile 有它自己的过程方法论的过程。

## 26.4.2 为 CMM 定位

CMM 是一个大家关注已久的话题，CMM 标准的提法颇为流行，下面笔者换个角度，根据软件工程概念模型的探讨，将 CMM 定位为“过程开发与改进的需求和测试方案”。

CMM 是软件过程开发的需求：

- 关键实践描述了应当“做什么”，而不是强制规定“如何做”；关键实践的描述就是过程开发的需求项；
- 关键实践被分组，成为一些关键过程域。相当于需求项的分组，便于管理；
- 关键过程域的实施都是为了达到一定的目的，从需求的层次角度（请参考 Wiegers 著陆丽娜译的《软件需求》一书），可将“目的”理解为“业务需求”，将关键过程域理解为“用户需求”，前者由后者来实现；
- CMM 通过定义的这些关键实践和关键过程域，覆盖了我们要开发的软件过程的主要目的（比如需求管理、产品工程）。

CMM 是软件过程开发的测试方案：

- 从原理上，需求就是测试依据。软件工程中的一条基本原理就是：依据需求进行测试；
- 从实施上，我们通常依据需求来编写测试用例，然后执行这些测试用例；
- Brain Marick 更是表述了他的具有革命性的观点：“我并不想写出一套用于捕捉用户愿望的需求，取而代之的是，我要写出一套测试，一旦这些测试能够通过，产品就能使她满意。”尽显大师风范；
- CMM 提供的大量提问单，和测试用例的概念对应得很完美，我们通过这些提问单就可以轻松测试出每一个关键实践进行得怎么样，进一步测试出每个关键过程域完成得如何，该组织的软件过程的能力成熟度有多高。

### 26.4.3 理解 RUP 定制

RUP 是著名的软件过程产品，包含了相当优秀的思想，比如：

- 为了使风险最小化，RUP 引入阶段概念和迭代开发模型，以便给开发者足够多的机会，在付出太多代价之前放弃或调整开发；
- 为了使并行最大化，RUP 引入工作流的概念，工作流是相关活动的集合，不仅工作流之间也是并行，工作流内部的活动也是可以并行的。

然而，根据具体的实践情况不同，使用 RUP 前应当对其进行适当定制。

“RUP 定制”属于“过程开发与改进”中的“过程开发”，而且严格来讲，是“过程开发的再工程”。再工程（reengineering）是对现有软件系统或软件过程的重新开发过程，包括逆向工程、新需求的考虑和正向工程三个步骤。

值得一提的是，“RUP 定制”既可以是“RUP 剪裁”，也可以是“RUP 扩充”。RUP 剪裁大家讨论的比较多了，有兴趣的读者可以阅读本书参考文献中所列的《随需而变的 RUP》一文。

著名的 RUP 扩充的例子有：

- Ronin International 公司将 RUP 扩充成为 EUP；
- 再比如爱立信在 RUP 的基础上进行扩充，开发出 ERUP 作为公司范围内的框架结构。

## 26.5 总结：软件工程概念模型的启示

### 26.5.1 软件工程，一门实践的科学

通过对软件工程概念模型的研究，强烈地感觉到，软件工程是一门实践的科学——它的出发点和最终目的是指导实践。基于此，我们至少应当注意两点：

- 从时间上，实践是发展的，基于实践的软件工程学科也必然是发展的，比如近几年，软件工程领域的发展就相当大。我们必须意识到这一点，不断学习新知识，才能适应软件实践的需要。Roger S. Pressman 说：“软件工程将发生变化——对此我们可以肯定。”
- 从内容上，软件实践的差异是巨大的，我们不能生搬硬套。正如 Alistair Cockburn 所说：“不同的项目需要不同的方法”。

### 26.5.2 软件过程，合适的才是最好的

据我所知，好的软件过程在不少人的脑子里，是一个“越……越……”的答案。比如，文档越多越好，分工越细越好，控制粒度越小越好，等等。还有，人们认为好的软件过程是可以照搬使用的，我听到过类似“我在某某大公司都是……”的抱怨。

其实通过软件工程概念模型的探讨，我们可以看到，软件过程是整个模型很多节点中的一个而已，这意味着软件过程和很多因素有着影响、被影响的关系：

- 软件类型、软件规模、软件重要程度、开发人员素质、管理和支持人员素质、合作单位素质等，都是影响软件过程制定的因素；
- 而且，且不可单纯认为软件过程仅仅涉及到开发人员，用户、合同确定者、投标者、项目管理者等，都可以成为软件过程的“涉众”；也就是说，他们都可能是待开发的软件过程的用户，应当收集他们对软件过程的要求。

### 26.5.3 对个人的启示

分析了软件工程概念模型，可以看出，从某种角度，可以认为软件产品是多种技术协作的结果。技术的协作最终表现为个人的协作，软件工程概念模型对个人有何启示呢？

- 明白自己知道的和不知道的，尊重他人，是一个团队的必要基础；
- 注重团队成员间技术的互补性和全面性。

### 26.5.4 呼唤高层次人才

看着模型，想着近年来国际上软件工程的巨大发展，深感国内在软件工程领域的落后，“透过变化看趋势、透过技术抓原理、把握软件工程发展脉搏”的高层次人才太少。

我辈尚需努力。



## 参考文献

---

1. Alistair Cockburn. 王雷等译. 编写有效用例. 机械工业出版社, 2002
2. Alistair Cockburn. 俞涓译. 敏捷软件开发. 人民邮电出版社, 2003
3. Andrew Hunt 等. 马维达等译. 程序员修炼之道——从小工到专家. 电子工业出版社, 2004
4. Andy Carmichael 等. 詹梅等译. 快速开发最佳软件. 电子工业出版社, 2004
5. Bruce Powel Douglass. 麦中凡等译. 实时设计模式——实时系统的强壮的、可扩展的体系结构. 北京航空航天大学出版社, 2004
6. Cay S.Horstmann 等. 程峰等译. Java 2 核心技术 (第 6 版) 卷 I: 基础知识. 机械工业出版社, 2003
7. Christine Hofmeister 等. 王千祥等译. 实用软件体系结构. 电子工业出版社, 2004
8. Colin Atkinson 等. 顾剑等译. 基于构件的产品线工程: UML 方法. 机械工业出版社, 2005
9. Connie U.Smith 等. 唐毅鸿等译. 软件性能工程. 机械工业出版社, 2003
10. Craig Larman. 姚淑珍等译. UML 和模式应用: 面向对象分析与设计导论. 机械工业出版社, 2002
11. Daniel J.Paulish. 白晓颖等译. 软件项目管理实用指南: 以体系结构为中心. 机械工业出版社, 2003
12. David M. Dikel 等. 张恂等译. 软件架构: 组织原则与模式. 机械工业出版社, 2002
13. Dean Leffingwell 等. 蒋慧等译. 软件需求管理: 统一方法. 机械工业出版社, 2002
14. Doug Rosenberg 等. 徐海等译. UML 用例驱动对象建模: 一种实践方法. 清华大学出版社, 2003
15. Douglas Schmidt 等. 张志祥等译. 面向模式的软件体系结构 卷 2: 用于并发和网络化对象的模式. 机械工业出版社, 2003
16. Edward R. Tufte. Envisioning Information. Graphics Press, 1990
17. Eric Evans. 陈大峰等译. 领域驱动设计——软件核心复杂性应对之道. 清华大学出版社, 2006
18. Erich Gamma 等. 李英军等译. 设计模式: 可复用面向对象软件的基础. 机械工业出版社, 2000

19. Frank Buschmann 等. 贲可荣等译. 面向模式的软件体系结构 卷 1: 模式系统. 机械工业出版社, 2003
20. Frederick P. Brooks. UMLChina 翻译组汪颖译. 人月神话. 清华大学出版社, 2002
21. Friedrich Steimann. Role = Interface: A Merger of Concepts.  
<http://www.adtmag.com/joop/article.asp?id=5123&mon=10&yr=2001>
22. Gary Pollice 等. 宋锐等译. 小型团队软件开发: 以 RUP 为中心的方法. 中国电力出版社, 2004
23. Grady Booch. 冯博琴等译. 面向对象分析与设计 (第 2 版). 机械工业出版社, 2003
24. Grady Booch. 邢春丽等译. 面向对象项目的解决方案. 机械工业出版社, 2003
25. Gunnar Overgaard 等. 任学群译. 用例: 模式与蓝图. 清华大学出版社, 2005
26. IBM. IBM Rational Unified Process 2003. <http://www.rational.com>, 2003
27. Ivar Jacobson 等. 程宾等译. 统一软件开发过程之路. 机械工业出版社, 2003
28. Ivar Jacobson 等. 徐锋译. AOSD 中文版. 电子工业出版社, 2005
29. Ivar Jacobson 等. 周伯生等译. 统一软件开发过程. 机械工业出版社, 2002
30. James Carey 等. 林星等译. 框架过程模式. 人民邮电出版社, 2003
31. James Rumbaugh 等. 姚淑珍等译. UML 参考手册. 机械工业出版社, 2001
32. Jeff Garland 等. 叶俊民等译. 大型软件体系结构: 使用 UML 实践指南. 电子工业出版社, 2004
33. Jim Highsmith. 姚旺生等译. 敏捷软件开发生态系统. 机械工业出版社, 2004
34. Karl E. Wiegers. 刘伟琴等译. 软件需求 (第 2 版). 清华大学出版社, 2004
35. Kent Beck. 唐东铭译. 解析极限编程. 人民邮电出版社, 2002
36. Kirk Knoernschild. 罗英伟等译. Java 设计: 对象、UML 和过程. 人民邮电出版社, 2003
37. Len Bass 等. 车立红译. 软件构架实践 (第 2 版). 清华大学出版社, 2004
38. Markus Völter 等. 徐异婕译. 模型驱动软件开发模式 (上). 非程序员, 第 50 期
39. Markus Völter 等. 徐异婕译. 模型驱动软件开发模式 (下). 非程序员, 第 51 期
40. Martin Fowler. 重构——改善既有代码的设计 (影印版). 中国电力出版社, 2003
41. Martin Fowler. 樊东平等译. 分析模式: 可复用的对象模型. 机械工业出版社, 2004
42. Martin Fowler. 王怀民等译. 企业应用架构模式. 机械工业出版社, 2004
43. Mary Shaw 等. 软件体系结构: 一门初露端倪学科的展望 (影印版). 清华大学出版社, 1999
44. Meliir Page-Jones. 包晓露等译. UML 面向对象设计基础. 人民邮电出版社, 2001
45. Mohamed E. Fayad 等. 姜晓红等译. 特定领域应用框架: 行业的框架体验. 电子工业出版社, 2004
46. Paul Clements 等. 张莉等译. 软件产品线实践与模式. 清华大学出版社, 2004
47. Paul Clements 等. 朱崇高译. 软件架构编档. 清华大学出版社, 2003
48. Per Kroll 等. 徐正生等译. Rational 统一过程: 实践者指南. 中国电力出版社, 2004

49. Peter Coad 等. Java Modeling In Color With UML: Enterprise Components and Process. Prentice Hall PTR, 1999
50. Peter Coad 等. 唐毅宏译. 对象模型: 策略 模式 应用 (第二版). 科学出版社, 2005
51. Peter Herzum 等. Business Component Factory. John Wiley & Sons, Inc., 1999
52. Philippe Kruchten. Common Misconceptions about Software Architecture. The Rational Edge, 2001
53. Philippe Kruchten. 周伯生等译. Rational 统一过程引论 (原书第 2 版). 机械工业出版社, 2002
54. R. J. A. Buhr 等. 用于面向对象系统开发的使用实例图 (影印版). 清华大学出版社, 1998
55. Robert Biddle. Caorui 译. 利用角色扮演和用例卡片进行需求复审. 非程序员, 第 11 期
56. Robert C. Martin. 邓辉译. 敏捷软件开发: 原则、模式与实践. 清华大学出版社, 2003
57. Roger S. Pressman. 梅宏译. 软件工程: 实践者的研究方法 (第 5 版). 机械工业出版社, 2002
58. Ronald J. Norman. 周之英等译. 面向对象系统分析与设计. 清华大学出版社, 2000
59. SEI. Software Architecture for Software-Intensive Systems.  
<http://www.sei.cmu.edu/architecture/>
60. Scott W. Ambler. Enterprise Unified Process. <http://www.enterpriseunifiedprocess.info>
61. Serge Demeyer 等. 莫倩等译. 软件再造: 面向对象的软件再工程模式. 机械工业出版社, 2004
62. Stephen H. Kan. 吴明晖等译. 软件质量工程——度量与模型 (第二版). 电子工业出版社, 2004
63. Stephen R. Palmer 等. 熊焕宇等译. 特征驱动开发方法原理与实践. 机械工业出版社, 2003
64. Suzanne Robertson 等. 王海鹏译. 掌握需求过程. 人民邮电出版社, 2003
65. Timothy C. Lethbridge 等. 张红光等译. 面向对象软件工程. 机械工业出版社, 2003
66. W. McUmbert. Object-oriented programming: Role-based design. <http://www.cse.msu.edu>
67. Xin Chen. 温昱等译. 应用框架的设计与实现——.NET 平台. 电子工业出版社, 2005
68. 邓成飞等. 软件工程管理. 国防工业出版社, 2000
69. 冯冲等. 软件体系结构理论与实践. 人民邮电出版社, 2004
70. 刘润东. UML 对象设计与编程. 北京希望电子出版社, 2001
71. 美国 Aspatore Books 公司. 张会娥等译. 卓越 CTO. 中国水利水电出版社 2004
72. 欧阳璟. 重量与轻量之间——访 ICONIX 之父 Doug Rosenberg. 程序员. 2005 年 10 月
73. 邵维忠等. 面向对象的系统设计. 清华大学出版社, 2003
74. 王立福等. 软件工程——技术、方法与环境. 北京大学出版社, 1997
75. 王咏刚等. 凌波微步: 软件开发警戒案例集. 清华大学出版社, 2002
76. 温昱. 见山只是山 见水只是水——提升对继承的认识. CSDN 开发高手. 2003 年 11 月
77. 温昱. 逻辑架构和物理架构在架构设计中的应用. 程序员. 2007 年 1 月

78. 温昱. 浅谈模式的正交分类. 程序员. 2005 年 5 月
79. 温昱. 软件架构概念思辨. 程序员. 2006 年 10 月
80. 温昱. 设计不佳的代价——顺序耦合性 Bug 案例一则. CSDN 开发高手. 2003 年 12 月
81. 温昱. 图论思想与 UML 应用 (上). 程序员. 2004 年 12 月
82. 温昱. 图论思想与 UML 应用 (下). 程序员. 2005 年 1 月
83. 温昱. 拥抱变化: 敏捷设计从理论到实践. 程序员. 2004 年 11 月
84. 温昱. 运用 RUP 4+1 视图方法进行软件架构设计. IBM DeveloperWorks, 2006
85. 温昱. 运用设计模式设计 MIME 编码类. CSDN 开发高手. 2003 年 9 月
86. 徐异婕. 基于角色的设计与团队开发. 上海市浦东新区第二届学术年会论文集 (二〇〇四). 浦东电子出版社, 2005
87. 尤克滨. UML 应用建模实践过程. 机械工业出版社, 2003
88. 张友生. 软件体系结构. 清华大学出版社, 2004
89. 郑人杰等. 基于软件能力成熟度模型 (CMM) 的软件过程改进——方法与实施. 清华大学出版社, 2003